Б. К. Мартыненко

СИНТАКСИЧЕСКИ УПРАВЛЯЕМАЯ ОБРАБОТКА ДАННЫХ

ИЗДАТЕЛЬСТВО САНКТ-ПЕТЕРБУРГСКОГО УНИВЕРСИТЕТА

	Состояние 5= {69}		
d	InitInt;SetDig	2	2
+	PushMonOp	2	3
–	PushMonOp	2	4
(	PushOpenPar	2	5
	Состояние 6= {31}		
d	InitInt;SetDig		12
	Состояние 7= {46}		
d	SetSign;InitInt;SetDig		13
+	SetSign		14
–	SetSign		15
	Состояние 12 = {36}		
=	AppDig;SetFrPart;AppFrPart;PushOpd		Sup
d	AppDig;SetDig		12
E	AppDig;SetFrPart;AppFrPart		7
+	AppDig;SetFrPart;AppFrPart;PushOpd;PushDyadicOp		8
–	AppDig;SetFrPart;AppFrPart;PushOpd;PushDyadicOp		9
*	AppDig;SetFrPart;AppFrPart;PushOpd;PushDyadicOp		10
/	AppDig;SetFrPart;AppFrPart;PushOpd;PushDyadicOp		11
	Состояние 13 = {37}		
=	AppDig;SetExp;AppExpPart;PushOpd		Sup
d	AppDig;SetDig		13
+	AppDig;SetExp;AppExpPart;PushOpd;PushDyadicOp		8

2004

С.-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Б. К. Мартыненко

СИНТАКСИЧЕСКИ УПРАВЛЯЕМАЯ ОБРАБОТКА ДАННЫХ

2-е издание, исправленное
и дополненное

ИЗДАТЕЛЬСТВО С.-ПЕТЕРБУРГСКОГО УНИВЕРСИТЕТА
2004

26

УДК 519.685.3
ББК 32.81
М25

Р е ц е н з е н т ы : д–р физ.-мат. наук, проф. *С.Н. Баранов* (С.-Петербургский филиал ЗАО "Моторола ЗАО"),
канд. техн. наук, доц. *В.П. Котляров* (С.-Петербургский техн. ун-т)

Печатается по постановлению
Редакционно-издательского совета
Санкт-Петербургского государственного университета

М а р т ы н е н к о Б.К.

М 25 **Синтаксически управляемая обработка данных.** Изд. 2-е, дополн. — СПб: Изд-во С.-Петербургского университета, 2004 г.
ISBN 5-288-03641-1

В монографии описывается актуальная для практической информатики технология синтаксически управляемой обработки данных, использующая кусочно-регулярную аппроксимацию КС-языков. Трансляции специфицируются при помощи RBNF-грамматик и реализуются посредством контекстно чувствительных сплайновых языковых процессоров. Технология применяется для решения синтаксических проблем, а также поддерживает объектно-синтаксическую парадигму программирования, которая выражается метафорической формулой "программа = объекты + грамматика".

Книга предназначена для специалистов, работающих в области теории и технологии программирования, а также для студентов, изучающих информатику.

Библиогр. 19 назв. Табл. 33. Ил. 83.

ББК 32.81

ISBN 5-288-03641-1

ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ К ПЕРВОМУ ИЗДАНИЮ

Эта книга является итогом многолетней работы "виртуального" научного коллектива по разработке и развитию технологии синтаксически управляемой обработки данных, которая проводилась в Ленинградском, а ныне — Санкт-Петербургском, университете с конца 1968 г. Средний возраст его участников — 20 лет, поскольку основу его всегда составляли и сейчас составляют студенты математико-механического факультета СПбГУ.

Мысль заняться технологией трансляции возникла у автора во время научной стажировки в A/S Regnecentralen (Копенгаген, 1967–1968 г.г.) под впечатлением от системы программирования GIER ALGOL 4 (для версии языка Алгол 60), созданной коллективом разработчиков под руководством проф. П. Наура. Каждый из девяти просмотров компилятора GIER ALGOL 4, некоторые из которых — обратные, представляет собой программу, вся логика которой упаковывается в компактную управляющую таблицу, открывающую доступ к действиям над данными. Казалось, что такие таблицы могли быть получены посредством некоторой технологии автоматически. Но, как выяснилось из разговора с Йорном Йенсеном — одним из основных участников проекта компилятора GIER ALGOL 4, никакой автоматизации не существовало, и все управляющие таблицы строились вручную посредством естественных рассуждений, основывающихся на представлениях разработчиков о синтаксисе входного языка и промежуточных языков этого компилятора.

Другим стимулом к разработке технологии трансляции явился заказ Научно-исследовательского центра электронной вычислительной техники Министерства радиопромышленности СССР на реализацию алгоритмического языка Алгол 68 для машин ЕС ЭВМ. Специфика условий работы состояла в том, что в тот период[1] язык Алгол 68 еще не был "заморожен". Многочисленные изменения появлялись чуть ли не в каждом очередном номере Алгол-бюллетеня и их следовало оперативно учитывать.

Истоки идей, положенных в основу технологии, связаны с именем Г.С. Цейтина, который указал на возможность использования регулярных выражений в правых частях правил КС-грамматики и определил способ их представления в виде ориентированных графов. Им было намечено основное направление эквивалентных преобразований грамматики: исключение крайних рекурсий за счет подстановок и введения итераций. Он показал, как можно построить конечный автомат по графу, представляющему некоторое регулярное выражение, операндами которого являются терминалы грамматики. Основываясь на этих предпосылках, первоначальный образ языкового процессора — основного элемента реализации трансляции — складывался по конечно-

[1] 1968–1974 годы.

автоматной модели распознавателя регулярного языка, а система проектирования — по соответствующему способу его построения.

Впоследствии были найдены ограничения, накладываемые на КС-грамматику, представленную в форме многокомпонентного графа, гарантирующие существование адекватного детерминированного магазинного анализатора, вид которого к тому моменту также был уточнен. Он совершенно не походил на анализаторы, моделирующие в своем магазине левосторонние (как LL-анализаторы) или правосторонние (как LR-анализаторы) выводы входной цепочки. Он напоминал магазинный автомат, но отличался от него тем, что движения его определялись либо по состоянию и входному символу (движения типа 1), либо по состоянию и верхнему символу магазина (движения типа 2).

Движение типа 1 состоит в записи в магазин некоторой цепочки символов над текущей вершиной магазина и переходе в следующее состояние управления с продвижением к следующему входному символу. Движение типа 2 использует верхний символ магазина для определения возвратного состояния, но при этом текущий входной символ не изменяется. По существу такой анализатор, отслеживая параллельным образом возможные маршруты порождения просканированной части входной цепочки, фиксирует их в своих состояниях управления. По мере дальнейшего сканирования входной цепочки какие-то из маршрутов отпадают, но регистрация их начальных участков в последовательности пройденных состояний уже не может быть отменена. Поэтому для завершения анализа, целью которого является нахождение маршрута порождения входной цепочки, используется обратный просмотр всей последовательности состояний. Такой анализатор воспроизводит метод Эрли с ограничениями, позволяющими реализовать его в компилятивной форме, т.е. с предвычислением прогнозирующих множеств исключительно по грамматике (безотносительно к какой-либо входной цепочке).

В дальнейшем анализатор стал рассматриваться как управляющий механизм для инициирования действий, составляющих процесс трансляции, а еще позже в его систему управления была введена контекстная чувствительность. Соответственно расширялся метаязык для спецификации трансляций и развивалась методика их реализации.

Важным этапом в развитии технологии была разработка метода оптимизации управляющего процессора, который дает, как правило, существенную экономию памяти[2], затрачиваемой на хранение управляющих таблиц.

Необходимость диагностирования ошибок, обнаруживаемых во время анализа входной цепочки, потребовала обдумывания также автоматической генерации диагностических сообщений и механизма их использования анализатором. С одной стороны, это привело к появлению синонимов — грамматических терминов, используемых для вербализации сообщений, и вспомогательных понятий как средства сохранения следов первоначальной синтаксической струк-

[2] Наблюдались, например, случаи экономии памяти от 25% (Algol 68) до 200–300% (Object Pascal).

туры входного языка, исчезающей в процессе эквивалентных преобразований грамматики, а с другой — в технологический комплекс был встроен генератор диагностических сообщений и редактор для придания им более естественной для человека формы.

В свой черед возник вопрос и о средствах для отладки генерируемых процессоров. Графовая форма представления грамматик оказалась столь же естественной для постановки на ней задачи автоматической генерации тестов, как и задачи анализа. И действительно, небольшая модификация компонент граф-грамматики, превращающая ее в сеть, позволяет свести задачу генерации оптимального бесконтекстного теста к сетевой задаче нахождения максимального потока при минимальной стоимости (или к классической задаче о китайском почтальоне), которая на такого рода сетях всегда имеет решение.

Наконец, была разработана общая архитектура SYNTAX-приложений, определены способы их сборки из типовых элементов (транслитераторов, конечных, сплайновых и челночных процессоров) и создана комфортабельная среда для наблюдения за их работой.

Отдельные компоненты технологического комплекса модифицировались по мере развития и совершенствования технологии. За это время некоторые из них сменили несколько платформ, начиная с реализаций на польской Одре-1204 (Алгол 60), ЕС ЭВМ (Алгол 68), терминальных станциях ТС-7063 (Форт) и кончая IBM PC (Паскаль).

Первыми участниками разработки первоначальной версии методики синтаксического анализа были научный сотрудник лаборатории системного программирования Вычислительного центра ЛГУ И.Б. Гиндыш и студент кафедры математического обеспечения ЭВМ Андраш Шоймоши, который через несколько лет после защиты дипломной работы в Ленинградском университете по этой же тематике успешно защитил диссертацию на степень доктора философии в Эрлангенском университете (Германия). Благодаря программам, созданным И.Б. Гиндышем, появилась уверенность в том, что на новую методику можно положиться при решении задачи синтаксического анализа такого сложного языка, как Алгол 68.

В последующие годы технологический комплекс SYNTAX развивался, главным образом, силами нескольких поколений студентов, среди которых были А.П. Попов (впоследствии применивший эту технологию при реализации Ады), А.А. Фрид (разработавший генератор и редактор диагностических сообщений на ЕС ЭВМ и IBM PC и спроектировавший интерфейс пользователя для подсистемы проектирования на IBM PC), Т.А. Фрид (запрограммировавшая генератор управляющих граф-схем на IBM PC), И.Б. Оськин (спроектировавший подсистему процессирования), Д.Н. Чистяков (построивший генератор оптимальных тестов), В.А. Петров (написавший программу преобразователя управляющих таблиц в таблицы решений и разработавший порождающий процессор, который реализует парадигму программирования "программа = грамматика + объекты"). М.В. Быховцева включила в технологический комплекс средства проверки приведенности трансляционной RBNF-грамматики. Н.В. Борейко

запрограммировала построитель управляющих таблиц обратного просмотра и предусмотрела включение в них дополнительной информации для экспозиции синтаксической структуры входных предложений. Р.А. Семизаров запрограммировал минимизатор управляющих таблиц обратного просмотра и реализовал обратный просмотр. Свой след в работе оставили также аспиранты Л.Н. Федорченко, И.Э. Косинец и А.И. Трощило.

Автор тоже выполнял "черную" работу, программируя интерпретативную версию анализатора на Одре-1204 (Алгол 60), весь прототип технологического комплекса (ТК) SYNTAX на ЕС ЭВМ (Алгол 68), генератор и оптимизатор управляющих таблиц прямого просмотра, а также реализацию абстрактных типов данных, используемых в ТК (Паскаль на IBM PC). На его долю также выпала задача обеспечивать преемственность в работе участников всех поколений.

Можно сказать с уверенностью, что концепции, на которых базируется SYNTAX-технология, успешно выдержали длительные испытания при реализации нескольких языков программирования, таких синтаксически сложных, как, например, Алгол 68, Ада и Object Pascal, при построении языковых конвертеров, как, например, Reduce 2 $\rightarrow$ Matematica, и в студенческих упражнениях. Технологический комплекс SYNTAX оказался весьма полезным в учебном процессе: на семинаре по технологии трансляции он используется для выполнения лабораторных работ, нацеленных на углубленное изучение принципов синтаксически управляемой обработки данных.

Специфику SYNTAX-технологии составляют следующие основные особенности:

- кусочно-регулярная аппроксимация входного языка трансляции (регулярные сплайны);
- сплайновые процессоры в качестве адекватного механизма реализации трансляций, которые решают задачи анализа, относящиеся к классу детерминированных МП-автоматов с эффективностью, свойственной конечным автоматам;
- четкое разделение синтаксического и семантического уровня спецификации и реализации трансляции, связь между которыми реализуется посредством контекстных символов (семантик и резольверов);
- контекстная чувствительность конечного управления сплайнового процессора (влияние текущего состояния операционной среды, в которой работает сплайновый процессор, на его управление);
- порождающие контекстно чувствительные сплайновые процессоры как основа объектно-синтаксического программирования, архитектура программ которого описывается метафорической формулой вида

"Программа = Объекты + Грамматика",

выражающей тот факт, что управляющая структура программы, заданная грамматикой и воплощенная в управляющей таблице, отделена от структуры данных и методов их обработки, определенных описанием операционной среды и воплощенных в библиотеке динамической линковки;

• возможность сборки средств синтаксически управляемой обработки данных из нескольких сплайновых процессоров.

Книга состоит из Введения, в котором кратко перечислены характерные черты и возможности SYNTAX-технологии, девяти глав, содержащих развернутое ее описание, краткого Заключения и Приложения, описывающего состав технологического комплекса SYNTAX, его функциональное наполнение и интерфейс пользователя. Указатель литературы содержит ссылки на публикации, так или иначе использованные в работе. Кроме того, книга снабжена Предметным указателем, который позволит читателю быстро найти конкретный материал.

На стиль изложения и выбор примеров в значительной степени повлияла студенческая аудитория, в которой материал этой книги излагался в течение нескольких лет в одноименном спецкурсе на отделении информатики математико-механического факультета Санкт-Петербургского государственного университета.

Я весьма признателен С.С. Сурину, который прочитал гл. 9 и сделал ряд ценных замечаний, и И.Б. Оськину за участие в подготовке Приложения.

Критические и в то же время благожелательные замечания рецензентов — С.Н. Баранова и В.П. Котлярова — в немалой степени повлияли на окончательный вариант этой книги. Им я выражаю свою искреннюю признательность.

Считаю своим приятным долгом поблагодарить всех многочисленных членов "SYNTAX-клуба" за деятельное участие в его работе; сотрудников лаборатории системного программирования ВЦ СПбГУ и малого ГП ТЕРКОМ, а также его генерального директора А.Н. Терехова за долголетнее добросердечное сотрудничество, поддержку работ по развитию технологического комплекса SYNTAX, а также за предоставление технических средств для подготовки рукописи этой книги.

Разумеется, ответственность за все недостатки этой книги несет исключительно ее автор.

Старый Петергоф,
1997 г.

ПРЕДИСЛОВИЕ КО ВТОРОМУ ИЗДАНИЮ

Первое издание книги вышло в самом конце 1997 г. и имело очень небольшой тираж, который быстро разошелся, так что даже факультетская библиотека не успела приобрести ни одного экземпляра. Поскольку эта книга используется в качестве учебного пособия студентами отделения информатики, слушающими одноименный спецкурс и участвующими в спецсеминаре "Технология трансляции", то было решено переиздать ее еще раз.

Данное издание книги отличается от предыдущего немногим.

В главу 1 включено два дополнительных рисунка (1.6 и 1.7), изображающих управляющую граф-схему синтаксически управляемого компилятора CALC в форме синтаксических диаграмм Н. Вирта, построенных с помощью новой функциональной компоненты технологического комплекса SYNTAX для операционной системы MS WINDOWS.

В главе 5 добавлен еще один параграф “5.4. Редукция КС-грамматик”. В нем обсуждается использование информации об эквивалентностях состояний и входных символов, получаемой при оптимизации процессора прямого просмотра, для минимизации граф-схемы и отображения ее в редуцированную RBNF-грамматику. Показывается, что задачу исключения несамовложенных нетерминалов из КС-грамматики можно решить, не прибегая к специальному алгоритму 5.1, использующем метод Гаусса, а полагаясь на механизм макроподстановок вспомогательных понятий при построении граф-схемы. Поясняется на примерах, почему для минимальной граф-схемы, построенной по грамматике чисел языка Алгол 68, не существует адекватная RBNF-грамматика, а для чисел языка Паскаль — существует. На простой модели языка арифметических выражений демонстрируется метод исключения всех нетерминалов из грамматики этого языка, несмотря на то, что все они, за исключением начального нетерминала, самовставленные.

Описание языка TSL пополнено деталями, касающимися использования порождающих грамматик (гл. 8); соответствующие дополнения внесены в Приложение (разд. "Подсистема процессирования"), в частности добавлен разд. 4: "Иллюстация парадигмы объектно-синтаксического программирования", включающий спецификацию регулярного фрактала "Снежинка" Коха с сопутствующими управляющими граф-схемой и управляющей таблицей, а также результатом — изображением фрактала на экране монитора.

Иллюстрации пополнены фотографией группы разработчиков компилятора GIER ALGOL (Алгол 60) из фирмы A/S Regnecentralen (Копенгаген, Дания), в которой автор в 1997–1998 гг. проходил научную стажировку.

Наконец, Предметный указатель перестроен заново.

Как всегда, в развитии ТК SYNTAX участвовали студенты и аспиранты кафедры информатики, среди которых считаю своим долгом отметить Е. А. Вергизову, С. А. Гришина, И. Н. Наумова и А. С. Лукичёва.

Старый Петергоф,
2004 г.

Введение

ОБЗОР ОСНОВНЫХ КОНЦЕПЦИЙ И ВОЗМОЖНОСТЕЙ SYNTAX-ТЕХНОЛОГИИ

Понятие *синтаксически управляемой обработки данных* подразумевает, что управление процессом обработки данных задается при помощи некоторой синтаксической структуры. В типичных приложениях, таких, как трансляция языков программирования, в которых этот принцип давно и успешно используется, управление процессом трансляции определяется синтаксической структурой предложений входного языка. Примером более изощренного применения принципа синтаксического управления является непосредственное задание структуры некоторого вычисления. В этом случае управляющая структура относится скорее не к исходным данным, а к состояниям вычислительного процесса.

1. Задание управляющих структур

Управляющие структуры традиционно описываются посредством грамматик. Чаще всего для этой цели используются контекстно-свободные и автоматные грамматики. В более развитых технологиях применяются модификации этих классов грамматик, такие, как атрибутные или аффиксные грамматики, позволяющие обогащать бесконтекстные синтаксические структуры контекстной информацией. В качестве основы чаще всего используются специальные подклассы КС-грамматик, обладающие "хорошими" технологическими свойствами.

Практиками ценятся такие свойства классов грамматик, как возможность определять достаточное многообразие свойств языков, их алгоритмическая распознаваемость, существование эффективного (с практической точки зрения) класса анализаторов языков, порождаемых грамматиками соответствующего класса, наличие метода для построения анализаторов и т.д.

В* SYNTAX-*технологии для спецификации управляющих структур и, в частности, языков используются так называемые детерминированные трансляционные* RBNF-*грамматики, которые обладают всеми перечисленными свойствами.

В терминологии SYNTAX-технологии трансляционные RBNF-грамматики являются средством спецификации трансляций. Они определяют не только бесконтекстную структуру входного языка, но также контекстные его особенности и (операционную) семантику. В них содержится достаточно информации для автоматической генерации адекватных процессоров, диагностирующих ошибки периода обработки, а также тестов для их испытания.

2. Трансляционные RBNF-грамматики

Трансляционные RBNF-грамматики являются двухуровневым формализмом для спецификации трансляций, который состоит из управляющей RBNF-грамматики, представляющей синтаксический уровень спецификации, и описания операционной среды, представляющего ее семантический уровень.

Управляющая RBNF-грамматика отличается от обычной BNF-грамматики[3] тем, что в ней помимо алфавитов нетерминалов и терминалов имеются два дополнительных алфавита *контекстных* символов: *семантических* и *резольверных*, а правые части правил представляют собой регулярные выражения над символами всех алфавитов грамматики. Эти выражения трактуются как формулы с регулярными операциями над множествами цепочек, составленных из терминальных и контекстных символов. При этом считается, что терминаль-ный или контекстный символ, используемый в качестве операнда регулярного выражения, представляет множество из одной односимвольной цепочки, а нетерминальный символ — множество[4] всех цепочек, представляющее его значение.

Таким образом, множество правил RBNF-грамматики рассматривается как своего рода "алгебраическая" система[5], неявно определяющая значения всех нетерминалов в качестве ее неизвестных, в то время как все прочие символы, точнее одноэлементные множества, ими представляемые, играют роль ее коэффициентов.

Множество цепочек, являющееся значением начального нетерминала, называется *синтаксическим управлением*, порождаемым данной управляющей RBNF-грамматикой, а каждая цепочка — элемент этого множества, — *управляющей цепочкой.* Если в управляющих цепочках игнорировать контекстные символы, мы получим бесконтекстную составляющую входного языка[6].

С другой стороны, интерпретация контекстных символов с каждым семантическим символом ассоциирует некоторое преобразование состояния операционной среды, а с резольверным символом — некоторый предикат, заданный на множестве состояний операционной среды. Интерпретация управляющей цепочки состоит в исполнении преобразований, ассоциированных с составляющими ее семантическими символами, при условии, что предикаты, ассоциированные с ее резольверными символами, выполняются[7]. Таким образом, предикаты "следят" за выполнением контекстных условий во входном предложении, а семантические преобразования воплощают его "смысл".

Трансляционная грамматика специфицирует трансляцию как некоторое отношение между предложениями входного языка и состояниями операционной среды. Считается, что входной язык состоит лишь из тех терминальных цепочек, которые входят в состав управляющих цепочек, порождаемых управляю-

---

[3] Обычная КС-грамматика, представленная в форме Бэкуса–Наура.

[4] Возможно бесконечное.

[5] В общем случае — нелинейная.

[6] Некоторый КС-надъязык над входным языком.

[7] Порядок синхронизации преобразований с вычислением предикатов определяется в разд. 1.2.

щей RBNF-грамматикой, и удовлетворяют контекстным условиям, определяемым предикатами, ассоциированными с составляющими их резольверными символами.

***Для записи трансляционных грамматик в* SYNTAX-*технологии используется специальный метаязык* (TSL[8]).** Описанию этого языка посвящена гл. 8.

3. Сплайновые процессоры как средство реализации трансляций

Для реализации трансляций в SYNTAX-технологии применяются *сплайновые процессоры*. Если резольверные символы не используются, то они аналогичны обычным детерминированным конечным или магазинным преобразователям, но обладают следующими особенностями.

Во-первых, семантические символы являются аналогом выходных символов, которые, однако, не пишутся на выходную ленту, а вызывают исполнение соответствующих преобразований операционной среды.

Во-вторых, управляющий элемент определяется в зависимости от текущего состояния управления и входного символа[9] (как в конечном преобразователе) или в зависимости от текущего состояния управления и магазинного символа[10], но никогда от трех параметров одновременно. При этом некоторые движения сопровождаются записью магазинной цепочки над текущим верхним символом магазина, а не вместо верхнего символа магазина, как в классическом МП-автомате. Запись магазинного символа соответствует приостановке текущего конечного процессора и запуску вспомогательного конечного процессора. ε-Движение использует верхний символ магазина для возврата в тот конечный процессор, работа которого была ранее приостановлена. Таким образом, сплайновый процессор играет роль "дирижера", руководящего совместной работой множества конечных процессоров, каждый из которых обрабатывает свой регулярный фрагмент входного языка. Именно за эту особенность процессорам и дано название сплайновых[11].

При использовании резольверных символов управляющие элементы выбираются еще и в зависимости от состояния контекста (операционной среды), тестируемого соответствующими предикатами. Это делает сплайновые процессоры контекстно чувствительными.

Подобно трансляционной грамматике, сплайновый процессор также состо-ит из двух компонентов — *управляющего процессора* и *операционной среды*. Управляющий процессор, сканируя входную цепочку, неявно реконструирует соответствующую управляющую цепочку и инициирует вызовы семантических процедур и предикатных функций, ассоциированных с её контекстными

[8] A Translation Specification Language.

[9] Непустое движение

[10] ε-движение.

[11] Если входной язык регулярен, то он аппроксимируется однофрагментным сплайном.

символами. Достигнутое посредством этих процедур состояние операционной среды рассматривается как результат трансляции.

Технологический комплекс* SYNTAX *включает универсальный управляющий процессор и средства для его настройки на реализацию конкретной трансляции за счет поставки соответствующей управляющей таблицы и операционной среды.

Определение сплайнового процессора, трансляции, им реализуемой, и описание способа построения управляющих таблиц можно найти в гл. 3.

4. Регулярные сплайны

Методику спецификации и реализации трансляций, применяемую в SYNTAX-технологии, уместно назвать термином *регулярные сплайны*, поскольку она фактически основана на идее аппроксимации КС-языка регулярными множествами подобно тому, как в методах приближенных вычислений функции на разных участках аппроксимируются различными полиномами. Это находит свое отражение и в том, что средства задания языков базируются на RBNF-грамматиках, правила которых определяют порождения для нетерминалов при помощи регулярных выражений, и в том, что сплайновый процессор на регулярных участках входной цепочки[12] ведет себя как конечный. Магазин же используется лишь для сопряжения разных конечных процессоров.

Другими словами, уместна метафора, согласно которой сплайновый процессор организует взаимодействие между множеством конечных процессоров, каждый из которых обрабатывает свой регулярный фрагмент языка.

В SYNTAX-технологии концепция регулярных сплайнов является ключевой для достижения высокой эффективности реализации трансляций.

5. Эквивалентные преобразования трансляционных грамматик

Разумеется, в общем случае за детерминированность магазинного процессора, используемого в SYNTAX-технологии, приходится расплачиваться дополнительными хлопотами. Известен алгоритм построения управляющего процессора, адекватного данной управляющей RBNF-грамматике. Но он приводит к успеху только в том случае, когда грамматика обладает определенными свойствами, гарантирующими детерминизм. Тестирование этих свойств встроено в алгоритм построения. В том случае, когда тестирование дает отрицательный результат, исходную трансляционную грамматику необходимо трансформировать так, чтобы она удовлетворяла необходимым условиям, но определяла ту же самую трансляцию.

Ясно, что трансляционную грамматику можно преобразовывать, сохраняя порождаемое ею синтаксическое управление[13] либо задавая эквивалентную ин-

[12] Регулярный участок входной цепочки — это такая ее часть, которая принадлежат одному регулярному множеству, распознаваемому некоторым конечным автоматом

[13] Такие преобразования назовем *сильно эквивалентными*.

терпретацию контекстных символов, либо преобразуя и то и другое совместно, заботясь лишь о неизменности трансляции[14].

Счастливым обстоятельством является то, что эквивалентные преобразования, имеющие целью достижение максимальной регуляризации грамматики[15] и, следовательно, максимальной эффективности процессора, одновременно ведут к достижению свойств грамматики, необходимых и достаточных для успешно-го построения процессора. Разумеется, безусловное достижение этих целей теорией не гарантируется.

Технологический комплекс* SYNTAX *предоставляет средства для тестирования свойств трансляционных грамматик и их эквивалентных преобразований.

Проблема эквивалентных преобразований подробно обсуждается в гл. 5.

6. Трансляционные граф-схемы

Алгоритм построения управляющего процессора фактически использует некоторое внутреннее представление управляющей грамматики в виде ориентированного помеченного псевдомультиграфа[16] специального вида, называемого *управляющей граф-схемой*.

Каждое правило управляющей RBNF-грамматики представляется в виде отдельной компоненты такого графа. Каждая компонента имеет две особые вершины. Одна из них — *начальная* — отличается тем, что из нее выходит некоторое множество дуг, но ни одна дуга в нее не входит. Она помечается меткой вида "begin-*A*", где *A* — определяемый нетерминал. Другая — *конечная*— характеризуется тем, что в нее входит некоторое число дуг, но ни одна дуга не выходит. Она помечена меткой вида "end-*A*". Внутренние вершины, помеченные терминальными и нетерминальными символами, являются образами соответствующих операндов регулярного выражения, составляющего правую часть правила грамматики.

Вершины соединяются ориентированными дугами, помеченными цепочка-ми контекстных символов, таким образом, чтобы следы маршрутов из началь-ной в конечную вершину данной компоненты образовывали регулярное мно-жество цепочек[17], описываемое регулярным выражением правой части правила, о котором идет речь. Как отмечалось, в общем случае компонента управляю-щей граф-схемы может содержать циклы и петли[18], а также иметь параллель-ные дуги, что является одним из признаков бесконтекстной неоднозначности управляющей грамматики.

---

[14] Такие преобразования назовем (просто) *эквивалентными*.

[15] Что фактически означает исключение максимально возможного числа нетерминалов.

[16] Допустимы параллельные дуги и петли.

[17] В общем случае эти цепочки состоят из символов всех алфавитов грамматики.

[18] Они представляют итерации.

В визуализированной форме управляющие граф-схемы есть не что иное, как синтаксические диаграммы Вирта, которые часто используются для описания синтаксиса языков программирования.

В SYNTAX-технологии управляющие граф-схемы используются как внутренняя форма представления RBNF-грамматик, удобная для построения управляющих таблиц процессоров и генерации оптимальных тестов для их тестирования. В [1] описано их непосредственное применение для анализа входного языка без построения управляющих таблиц (интерпретативный метод анализа). Соответственно их можно использовать для непосредственного управления процессированием.

Технологический комплекс* SYNTAX *обеспечивает автоматическое преобразование управляющих* RBNF-*грамматик в форму управляющих граф-схем.

Определение управляющей граф-схемы и метод ее построения по управляющей грамматике даются в разд. 1.5.

7. Челночные трансляции

Как уже отмечалось в разд.3, внутренняя задача процессора состоит в том, чтобы реконструировать управляющую цепочку, соответствующую данной входной, или отвергнуть её как ошибочную. В общем случае такая реконструкция выполняется в два этапа.

Сначала производится сканирование входной цепочки в прямом направлении (от начала к концу), при котором последовательность состояний процессора регистрирует множество возможных порождающих маршрутов в управляющей граф-схеме для каждой ее префиксной части. Затем эта последователь-ность состояний прямого просмотра сканируется в обратном порядке для того, чтобы выявить множество полных маршрутов порождения входной цепочки. При этом учитывается контекст, зафиксированный операционной средой. Эти маршруты начинаются в начальной вершине заглавной компоненты, т.е. той компоненты управляющей граф-схемы, которая представляет правило для начального нетерминала, и заканчиваются в конечной ее вершине. В общем случае они проходят через некоторые другие компоненты. Следы этих маршрутов определяют множество управляющих цепочек, соответствующих данной входной. При этом метки начальных и конечных вершин, вошедших в маршрут, представляют синтаксическую структуру (в скобочной записи) управляющей цепочки, порождаемой этим маршрутом.

Если ни одной такой управляющей цепочки не существует, входная цепочка *ошибочна.*

Если таких управляющих цепочек несколько, входная цепочка *синтаксически неоднозначна.* Метод допускает синтаксическую неоднозначность, если она не приводит к семантической неоднозначности. В этом случае семантические следы порождающих маршрутов должны быть идентичны.

Поскольку задача синтаксического анализа входной цепочки, состоящая в нахождении множества порождающих ее маршрутов, решается в два просмотра[19], естественно использовать обратный просмотр для продолжения преобразований операционной среды, начатых на прямом просмотре. Это приводит к понятию *челночной трансляции*, определяемому как отношение между входными цепочками и финальными состояниями операционной среды или некоторой ее части после выполнения обоих просмотров. Соответственно алфавиты контекстных символов также разделяются по просмотрам.

Различаются семантики прямого и обратного просмотров, а также резольверы прямого и обратного просмотров. Обратный просмотр, как и прямой, организован в форме процессора. Как уже отмечалось, на его вход поступают состояния прямого просмотра.

Технологический комплекс* SYNTAX *предоставляет средства для автоматического построения управляющих таблиц челночных процессоров.

Метод построения управляющих таблиц челночного процессора описывается в гл. 3.

8. Спецификация микролексики

В транслирующих системах входной текст первоначально поступает из некоторого входного потока (файла на диске или клавиатуре) и обрабатывается политерно. В качестве обработчика литер обычно используется конечный процессор, который определяет способ ее обработки в зависимости от того, к кАкому микролексическому классу она относится. Такими классами могут быть буквы, цифры, спецзнаки и т.п. В различных приложениях микролексика разная. Следовательно, необходимы средства для ее спецификации и реализации.

В структуре спецификации трансляции предусмотрен специальный раздел для определения микролексики. Он открывается ключевым словом MICROLEXICS. Спецификация микролексики для конкретного приложения обрабатывается встроенным в систему конечным процессором, формирующим микролексические классы, которые затем используются штатным транслитератором для выдачи микролексем взамен входных литер.

Описание средств спецификации микролексики дается в гл. 7.

9. Комплексирование процессоров

Итак, основными средствами реализации трансляций в SYNTAX-технологии являются однопросмотровые и челночные сплайновые процессоры. В сложных системах синтаксически управляемой обработки данных могут совместно использоваться несколько взаимодействующих процессоров разного типа. Например, в компилирующих системах конечный процессор может быть

[19] Заметим, что задача распознавания решается до конца прямым просмотром. Это значит, что если входная цепочка содержит синтаксические (бесконтекстные или контекстные) ошибки, то они обнаруживаются на прямом просмотре. Исключение составляют контекстные ошибки, которые выявляются резольверами обратного просмотра.

использован в качестве сканера входного текста, решающего задачу лексического анализа и поставляющего лексемы сплайновому процессору, выполняющему роль синтаксического анализатора.

В свою очередь, сканер нуждается в транслитераторе, поставляющем ему микролексемы — литеры с классифицирующими признаками, такими, как 'буква', 'цифра', 'операция', 'спецзнак' и т.п.

При разработке такого комплекса процессоров микролексика описывается в разделе MICROLEXICS спецификации сканера, лексика входного языка — в разделе SYNTAX той же спецификации, а его синтаксис — в разделе SYNTAX спецификации анализатора. Соответственно по описанию микролексики генерируются списки микролексических классов для стандартного транслитератора, по описанию лексики — лексический анализатор, а по описанию синтаксиса — сплайновый[20] , используемый в роли синтаксического анализатора.

В процессе трансляции синтаксический анализатор вызывает лексический анализатор всякий раз, как ему требуется очередная лексическая единица (лексема), который, в свою очередь, несколько раз вызывает транслитератор, чтобы получить несколько микролексем, из которых образуется очередная лексема.

Другой способ композиции разных процессоров состоит в использовании одного из них в качестве семантики другого.

Возможность разделять процесс синтаксически управляемой обработки данных на несколько уровней, разрабатываемых относительно независимо, с последующей комплексацией реализующих их процессоров в рамках одного средства СУОД[21], является практически ценной чертой SYNTAX-технологии.

Технологический комплекс* SYNTAX *располагает средствами конструирования систем обработки данных из нескольких процессоров различных типов.

10. Встроенные функции управляющего процессора

Как отмечалось, процессор, реализующий конкретную трансляцию, есть не что иное, как универсальный управляющий процессор, параметризованный управляющей таблицей и операционной средой. Этот универсальный процес-сор поддерживает ряд полезных стандартных функций, которые можно вызывать из семантик и резольверов. К ним относятся функции, открывающие доступ к входным и выходным лексическим буферам, функции управления види-мостью входных лексем на каждом уровне процессирования, а также средства обмена информацией между процессорами разных уровней (встроенный механизм сообщений).

Дополнительные функции, встроенные в управляющий процессор, обеспечивают благоприятные возможности для реализации разнообразных* SYNTAX*-приложений.

Описание стандартных встроенных функций можно найти в гл. 7.

[20] Простой или челночный.

[21] Синтаксически Управляемая Обработка Данных.

11. Генерация диагностических сообщений

В практически используемых системах обработки данных важное значение имеет контроль корректности исходных данных. В методологии синтаксически управляемой обработки данных это требование удовлетворяется как нечто само собой разумеющееся. Более того, грамматика, описывающая все множество возможных исходных данных, обеспечивает терминологический базис для формулирования сообщений об обнаруживаемых дефектах в исходных данных.

Технологический комплекс* SYNTAX *имеет средства автоматической генерации и редактирования диагностических сообщений, которые затем используются для характеристики ошибок на входе системы процессирования.

Описание этих средств можно найти в Приложении.

12. Генерация тестов

Теория и основанная на ней SYNTAX-технология гарантируют, что получаемое с их помощью средство синтаксически управляемой обработки данных полностью соответствует его TSL-спецификации. Однако на практике такой гарантии бывает недостаточно, поскольку нельзя утверждать, что сама исходная спецификация правильно отражает потребности заказчика. Единственно возможным средством проверки "неформальной" правильности получаемого средства обработки данных является его тестирование.

Представление управляющей грамматики в виде графа естественным образом наводит на мысль о родственности задачи генерации тестов задаче о "Китайском почтальоне" — классической задаче математического программирования, сетевой задаче нахождения максимального потока при минимальной стоимости. Действительно, чтобы гарантировать однократное исполнение каждой семантики, достаточно найти такое множество входных цепочек, порождающие маршруты которых покроют все дуги управляющей граф-схемы (критерий полноты C_1). Тогда гарантированно будут задействованы все возможные преобразования операционной среды, поскольку прохождение по дуге, распознаваемое управляющим процессором, инициирует вызов семантических процедур, интерпретирующих семантические символы, помечающие эту дугу. Параметр i критерия полноты C_i можно варьировать в зависимости от потребности получения более сильных тестов. Так, при $i=2$ тестом покрывается маршрут, который обладает тем свойством, что каждая дуга, входящая в данную вершину, сочетается с каждой дугой, выходящей из нее. Другими словами, этот критерий гарантирует испытание каждой возможной пары семантик. И вообще, критерий C_n обеспечивает испытание каждого возможного n-сочетания семантик.

Технологический комплекс* SYNTAX *в своем составе имеет генератор тестов, позволяющий автоматически строить полные минимальные тесты для любого заданного критерия тестирования C_n **(n = 1, 2, ...).**

Методика генерации тестов рассмотрена в гл. 9.

13. Спецификация вычислений

Как упоминалось в разд. 2, трансляционные RBNF-грамматики можно использовать для непосредственной спецификации вычислений произвольного типа. Действительно, посредством трансляционной грамматики можно задать вычислительный процесс, применяя резольверы как средство тестирования условий, в зависимости от выполнения или невыполнения которых выбирается дальнейший порядок вычислений, реализуемый посредством действий, ассоциируемых с терминальными и семантическими символами грамматики. Соответствующий (порождающий) процессор при таком его применении управляется таблицей, в которой главным является резольверный вход, а не входные символы, которые в данном случае трактуются как действия над данными, составляющими операционную среду. Управляющая таблица порождающего процессора является аналогом таблицы решений, а сам он вместе с таблицей подобен специализированной вычислительной машине, построенной для реализации одного класса вычислений. Фактически управляющая таблица порождающего процессора получается путем транспозиции входов управляющей таблицы анализирующего процессора.

14. Объектно-синтаксическое программирование

Практика использования порождающих процессоров для реализации вычислений привела к пониманию плодотворности сочетания синтаксических принципов управления с методологией объектно-ориентированного программмирования. В самом деле, операционную среду можно представить как некоторую коллекцию объектов, резольверы — как методы-предикаты, а терминалы — как методы-процедуры. Благодаря такому подходу мы имеем *раздельную спецификацию структуры управления* (в форме управляющей грамматики) *и данных с методами их обработки* (в форме описания операционной среды).

Архитектуру таким образом организованной программы коротко можно описать при помощи следующей метафорической формулы:

Программа = Объекты + Грамматика.

Она означает, что посредством грамматики данную коллекцию объектов можно реорганизовать для нового применения. С другой стороны, не изменяя структуры управления, можно изменить конкретное представление данных или методы их обработки. Конструктивно такая программа состоит из постоянного кода, функционирование которого задается сменными управляющими таблицами и коллекциями объектов, представленными в форме библиотек динамической линковки (DLL[22]) .

[22] Dinamic-link library.

Поскольку структура управления воплощается в управляющей таблице процессора, она может быть проанализирована, оптимизирована и компактно закодирована. SYNTAX-технология обеспечивает для этого надлежащие средства.

Итак, методология объектно-синтаксического программирования подразумевает использование "универсального" вычислителя (управляющего процессора), ориентированного на фиксированный формат управляющих таблиц. Он может быть настроен на конкретное применение путем поставки конкретной управляющей таблицы и объектов обработки.

Достоинства объектно-синтаксической парадигмы программирования подробно обсуждаются в гл. 2.

15. Структурная модульность

Синтаксический подход индуцирует еще одну разновидность модульности, которую уместно назвать *структурной*. И в самом деле, "программируя" в грамматиках, мы выражаем наш алгоритм в терминах программных конструкций (нетерминалов), определяемых соответствующими правилами грамматики. Когда мы пишем одно правило грамматики, мы определяем некоторую программную конструкцию в терминах действий, относящихся к самой данной конструкции (представленных вхождениями семантик или терминалов в порождающих грамматиках), и тех действий, которые реализуются составляющими ее подконструкциями (представленными вхождениями нетерминалов). Таким образом, используя SYNTAX-технологию при проектировании некоторого средства СУОД, мы можем поочередно сосредоточиваться на деталях реализации одной программной конструкции, временно не вникая в реализацию ее подконструкций.

16. Средства отладки средств синтаксически управляемой обработки данных

При отладке средств синтаксически управляемой обработки данных необходимо иметь возможность увидеть, как они работают на различных исходных данных.

Технологический комплекс* SYNTAX *обеспечивает комфортную обстановку для отладки разрабатываемого средства СУОД. В частности, он дает возможность трассировать его работу по выбору на уровне одного или нескольких составляющих его процессоров.

Эти возможности описываются в Приложении.

17. Языки спецификации операционного окружения

В первом выпуске комплекса SYNTAX в качестве языка спецификации операционной среды используется Borland Pascal 7.0. Для этого в интегрированную среду комплекса встроен штатный компилятор **bpc**, формирующий для каждого процессора его операционную среду в виде библиотеки динамической линковки.

Ценой небольших усилий в комплекс можно включить и другие компиляторы, формирующие DLL-библиотеки, например компиляторы для языка C или C^{++}. И тогда для спецификации операционной среды можно будет использовать альтернативные языки программирования. Более того, операционную среду разных процессоров можно было бы задавать на разных языках. Иначе говоря,

SYNTAX-*технология открыта по отношению к языкам описания операционной среды.*

18. Перспективы развития SYNTAX-технологии

В план работ по развитию технологического комплекса SYNTAX входят

- уточнение архитектуры средств СУОД как программных компонент, подлежащих встраиванию в другие приложения;
- подключение средств мультиязыковой спецификации операционной среды;
- разработка специализированных операционных сред для различных областей приложений SYNTAX-технологии;
- развитие средств эквивалентных преобразований трансляционных грамматик;
- включение в комплекс более совершенных редакторов и средств печати отчетных документов по проектам трансляций;
- совершенствование методики автоматической генерации тестов: построение метода генерации тестов по многокомпонентным управляющим граф-схемам, разработка методики генерации тестов, удовлетворяющих определенным контекстным условиям;
- замена интерфейса пользователя, имеющегося в настоящее время, более совершенным и удобным в работе;
- построение версии комплекса SYNTAX для использования его в среде MS WINDOWS;
- разработка механизма параметризации семантических процедур и предикатных функций.

Глава 1

СПЕЦИФИКАЦИЯ И РЕАЛИЗАЦИЯ ТРАНСЛЯЦИЙ

1.1. СПОСОБЫ СПЕЦИФИКАЦИИ И РЕАЛИЗАЦИИ ТРАНСЛЯЦИЙ В SYNTAX-ТЕХНОЛОГИИ

Любая информационная технология, базирующаяся на принципе синтаксически управляемой обработки данных, характеризуется способом спецификации трансляций, и эта характеристика находит свое отражение в используемых языках спецификации.

В SYNTAX-технологии для спецификации трансляций используется метаязык TSL, который является входным языком технологического комплекса SYNTAX, предназначенного для разработки средств синтаксически управляемой обработки данных. В применении к реализации языков программирования TSL служит для описания синтаксиса и семантики языков в форме трансляционных грамматик.

Трансляционная грамматика определяет как *бесконтекстную структуру* предложений входного языка, так и дополнительные — *контекстные — условия*, которым эти предложения должны удовлетворять. Она также определяет семантику входного языка в терминах *действий* некоторого *гипотетического языкового процессора над операционной средой.* Эти действия определяются в зависимости от бесконтекстной синтаксической структуры входного предложения с учетом контекстных условий. Трансляционная грамматика состоит из *управляющей грамматики* и *описания операционной среды*.

Управляющая грамматика — это контекстно свободная грамматика с RBNF-правилами, в которых помимо нетерминалов и терминалов можно ис-пользовать дополнительные *семантические* и *резольверные* символы. Эти символы мы будем называть *контекстными.*

Описание операционной среды определяет ее как некоторое пространство данных — элементов операционной среды, а интерпретацию контекстных символов — как множество преобразований и предикатов над текущим состоянием операционной среды (которое можно представить как точку в пространстве данных с координатами, обусловленными текущими значениями элементов операционной среды). А именно: с каждым из семантических символов ассоциируется некоторое преобразование состояния операционной среды, а с каждым резольверным символом — некоторый предикат, определенный в пространстве состояний операционной среды. Контекстные символы вместе с их интерпретациями будем называть соответственно *семантиками* и *резольверами.*

Строго говоря, фиксированными в TSL являются лишь средства для записи управляющих грамматик. Для описания же операционной среды можно использовать Borland Pascal 7.0 и другие языки программирования, поддерживающие механизм DLL-библиотек[23], например C или C^{++}.

В SYNTAX-технологии трансляционную грамматику можно трактовать как *анализирующую* и как *порождающую* некоторый язык.

Анализирующие трансляционная грамматика и процессор. При трактовке трансляционной грамматики как анализирующей строится *анализирующий процессор входного языка*, который ею определяется. Этот процессор в общем случае[24] подобен детерминированному магазинному преобразователю (рис.1.1), но имеет некоторые отличия.

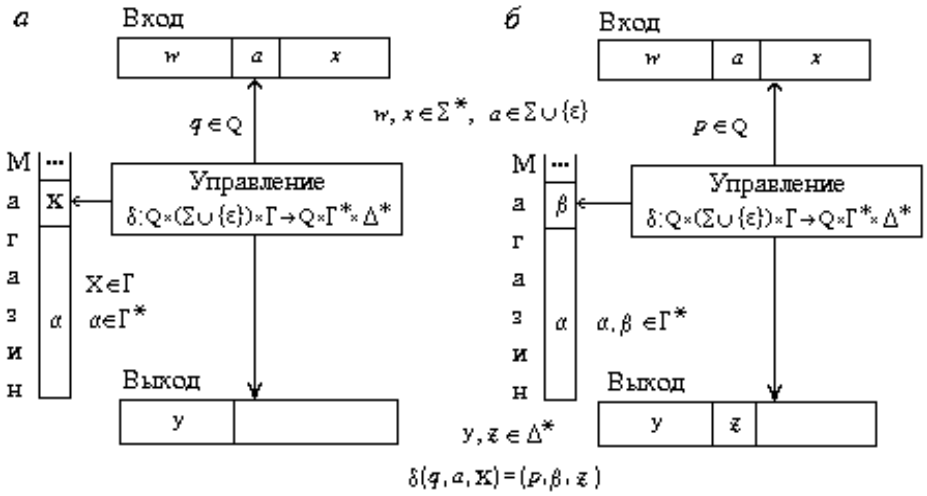


Рис.1.1. Магазинный преобразователь.
а — текущая конфигурация, *б* — следующая конфигурация.
Q — множество состояний управления, Σ — входной алфавит,
Γ — магазинный алфавит, Δ — выходной алфавит.

Во-первых, движение процессора зависит от текущих состояния управления и входного символа или от текущего состояния управления и верхнего символа магазина, но не от трех параметров сразу, как в традиционном магазинном преобразователе.

В первом случае происходит продвижение по входной цепочке к следующему символу (непустое движение), а во втором — стирание верхнего символа магазина (пустое или ε-движение). При использовании резольверов выбор

[23] В первой версии технологического комплекса SYNTAX можно использовать только Borland Pascal 7.0. Переход на другие языки программирования потребует сравнительно небольших усилий. Они связаны с перестройкой реформатизатора разделов описания операционной среды в TSL-спецификации, написанных на альтернативном языке программирования, в библиотечный модуль, оформленный по правилам альтернативного языка, и заменой в интегрированной среде комплекса компилятора **bpc** на командный компилятор другого языка.

[24] В случае регулярного входного языка процессор подобен конечному преобразователю.

движения зависит также и от текущего состояния операционной среды. Точнее, на каждом такте работы процессора в зависимости от текущего состояния управления, входного символа и состояния операционной среды, тестируемого предикатами, ассоциированными с резольверными символами, выбирается управляющий элемент, который определяет, какую цепочку поместить на вершину магазина (над верхним символом магазина), какие преобразования операционной среды выполнить и принять ли текущий входной символ. При этом переходное состояние определяется по-разному в зависимости от того, принимается ли текущий входной символ или нет. Если символ принимается (непустое движение, рис. 1.2[25]), то новое (переходное) состояние определяется непосредственно данным управляющим элементом. Если символ не принимается (пустое движение, рис. 1.3), то новое (возвратное) состояние определяется по текущему (подав ляемому) состоянию и верхнему символу магазина (символ в этом случае удаляется из него) в зависимости от текущего состояния операциионной среды.

Ошибка на входе определяется тем, что в текущей конфигурации очередной управляющий элемент не определен. При этом различаются *бесконтекстные* и *контекстные ошибки.* А именно: если текущий входной символ не приемлем ни при каком контексте, то имеет место бесконтекстная ошибка. Если текущий входной символ был бы принят при выполнении хотя бы одного из нескольких заданных управляющей таблицей условий, то имеет место контекстная ошибка.

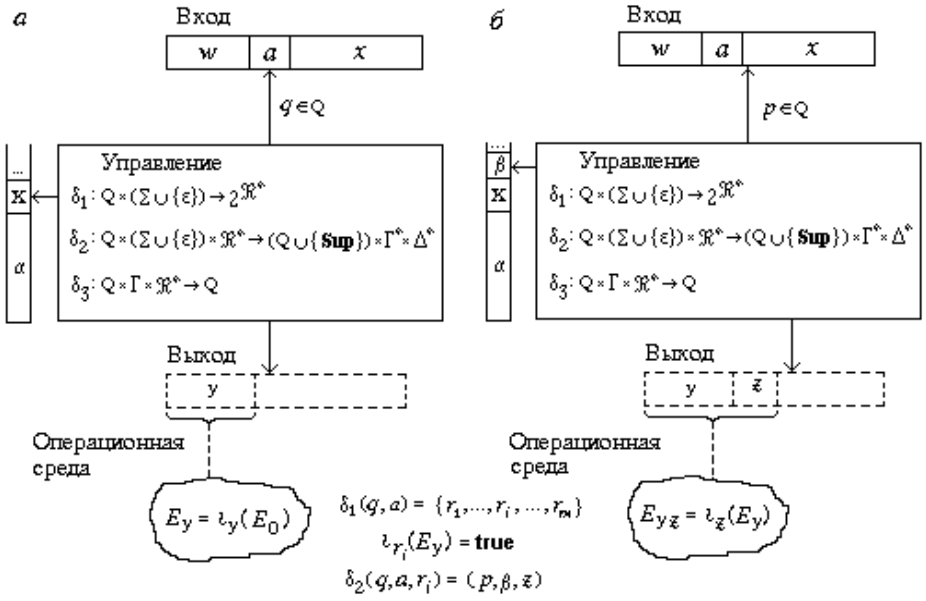


Рис. 1.2. Магазинный процессор (непустое движение).
а — текущая конфигурация, б — следующая конфигурация.
Здесь **Sup** — сокращение от **Suppress**.

[25] Пояснение обозначений, использованных на рис. 1.2, 1.3, а также точное определение анализирующего процессора и трансляции, им реализуемой, можно найти в разд. 1.3.

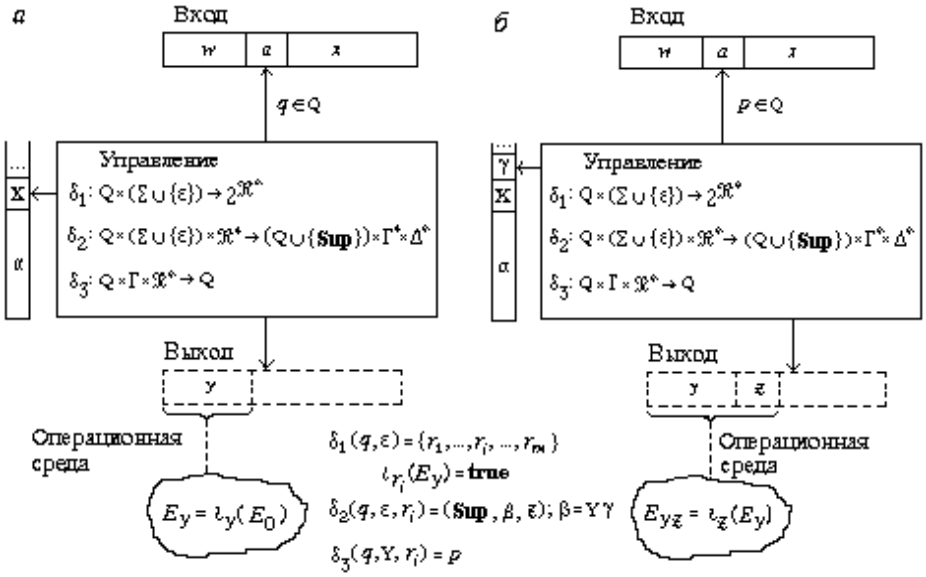


Рис. 1.3. Магазинный процессор (пустое движение).
а — текущая конфигурация, *б* — следующая конфигурация.

Процессор заканчивает свою работу, когда просканировано все входное предложение и достигнуто одно из конечных состояний управления при пустом магазине[26]. Финальное состояние операционной среды или некоторой ее части (например, содержимое некоторого файла) представляет "смысл" прочитанного входного предложения или, если угодно, его перевод.

Порождающие трансляционная грамматика и процессор. При трактовке трансляционной грамматики как порождающей адекватный ей процессор также действует как порождающий[27]. В этом случае он не имеет ничего на входе, а напротив, генерирует выход, причем выходные символы интерпретируются как некоторые преобразования состояний операционной среды. На каждом такте его работы в зависимости от текущих состояний управления и операционной среды, состояние которой тестируется резольверами, выбирается управляющий элемент, который определяет, нужно ли на выход выдавать какой-нибудь символ, и если нужно, то какой именно, что равносильно указанию, нужно ли выполнять какие-либо преобразования операционной среды, и если нужно, то какие именно, какую цепочку поместить в магазин и в какое состояние управления перейти. При этом новое состояние управления, как в анализирующем процессоре, зависит от того, задан ли символ, который требуется выдавать на выход, или нет. Если символ задан, то новое (переходное) состояние определяется самим управляющим элементом (непустое движение). Если символ не указан, то новое (возвратное) состояние зависит от текущего состояния управле-

[26] Строго говоря, в практических применениях первое из упомянутых условий не всегда соблюдается.

[27] Точное определение порождающего процессора см. в разд. 1.4.

ния, верхнего символа магазина и контекста, представляемого текущим состоянием операционной среды.

Что касается семантик, то их роль в порождающих процессорах — вспомогательная: они могут использоваться для реализации некоторых подготовительных действий. Что считать основными действиями, связываемыми с выходными символами, а что — вспомогательными, связываемыми с семантическими символами, зависит от предпочтений пользователя.

Трактовка трансляционных грамматик и процессоров как порождающих используется при задании вычислений с раздельным описанием структуры управления алгоритма и данных, им обрабатываемых. Преимущества такого подхода к обработке данных подробно обсуждаются в гл. 2.

Дадим точное определение понятий, положенных в основу технологии.

1.2. ТРАНСЛЯЦИОННАЯ ГРАММАТИКА

Трансляционная грамматика есть формальная система $G_t = (G_c, \mathscr{E})$, где G_c — управляющая (control) грамматика; $\mathscr{E}$ — описание операционной среды.

Управляющая грамматика определяет синтаксис входного языка, вернее, способ его обработки (трансляции) через синтаксическую структуру его предложений. Описание же операционной среды задает операционную (трансляционную) семантику входного языка.

Управляющая RBNF-грамматика. Формальная система $G_c=(N,T,\mathfrak{R},\Sigma,P,S)$, где N — словарь нетерминальных символов или нетерминалов; T — словарь терминальных символов или терминалов; $\mathfrak{R}$ — словарь резольверных символов или резольверов; Σ — словарь семантических символов или семантик; $P=\{A: \mathscr{R}_A. \mid A \in N, \ \mathscr{R}_A$ — регулярное выражение относительно символов из множества $N \cup T \cup \mathfrak{R} \cup \Sigma\}$ — множество правил, S — начальный нетерминал, называется *управляющей* RBNF-*грамматикой*.

Управляющая грамматика специфицирует синтаксическое управление — множество цепочек $\mathscr{C}(G_c)=\lambda(\mathscr{R}_S)$ над терминальными и контекстными символами (т. е. символами из множества $T \cup \mathfrak{R} \cup \Sigma$), где $\lambda(\mathscr{R})$ определяется в зависимости от вида $\mathscr{R}$ следующим образом:

$$\lambda(\mathscr{R}) = \begin{cases} \{a\}, & \text{если } \mathscr{R}=a,\ a \in T \cup \mathfrak{R} \cup \sum \text{ или } a=\varepsilon; \\ \lambda(\mathscr{R}_A), & \text{если } \mathscr{R}=A,\ A \in N, \text{ и существует правило } A: \ \mathscr{R}_A.; \\ \bigcup\limits_{k=0}^{\infty} (\lambda(\mathscr{R}_1))^k, & \text{если } \mathscr{R} = \mathscr{R}_1^*; \\ \bigcup\limits_{k=1}^{\infty} (\lambda(\mathscr{R}_1))^k, & \text{если } \mathscr{R} = \mathscr{R}_1^+; \\ \lambda(\mathscr{R}_1) \bigcup\limits_{k=1}^{\infty} (\lambda(\mathscr{R}_2)\lambda(\mathscr{R}_1))^k, & \text{если } \mathscr{R} = \mathscr{R}_1 * \mathscr{R}_2; \\ \lambda(\mathscr{R}_1)\lambda(\mathscr{R}_2), & \text{если } \mathscr{R} = \mathscr{R}_1,\ \mathscr{R}_2; \\ \lambda(\mathscr{R}_1) \cup \lambda(\mathscr{R}_2), & \text{если } \mathscr{R} = \mathscr{R}_1;\ \mathscr{R}_2; \\ \lambda(\mathscr{R}_1) \cup \{\varepsilon\}, & \text{если } \mathscr{R} = [\mathscr{R}_1]; \\ \lambda(\mathscr{R}_1), & \text{если } \mathscr{R} = (\mathscr{R}_1). \end{cases}$$

Здесь $\mathscr{R}_1$ и $\mathscr{R}_2$ — некоторые регулярные выражения; определение вида регулярного выражения производится с учетом старшинства операций и расстановки скобок, при этом предполагается, что наивысшее старшинство имеют унарные операции итерации или замыкания (* и $^+$), за ними следует итерация с разделителем ($*$), далее конкатенация (,) и, наконец, объединение (;).

Описание операционной среды. Формальная система $\mathscr{E} = (\mathrm{E}, \mathrm{H}, \mathscr{I}_\Re, \mathscr{I}_\Sigma, e_0)$, где E — *пространство состояний операционной среды* — область определения *предикатов*: $\mathscr{I}_\Re = \{\iota_\rho : \mathrm{E} \to \{\mathbf{false}, \mathbf{true}\} \mid \rho \in \Re\}$, ассоциированных с резольверными символами, и *преобразований операционной среды*: $\mathscr{I}_\Sigma = \{\iota_\sigma : \mathrm{E} \to \mathrm{E} \mid \sigma \in \Sigma\}$, ассоциированных с семантическими символами; H — *объектное подпространство* (та часть операционной среды, состояние которой представляет особый интерес); $e_0 \in \mathrm{E}$ — *начальное состояние операционной среды*, называется *описанием операционной среды*.

Трансляция, определяемая трансляционной грамматикой. Пусть $e \in \mathrm{E}$ — некоторое состояние операционной среды, $\rho \in \Re^*$ — некоторая резольверная цепочка, а $\sigma \in \Sigma^*$ — некоторая семантическая цепочка.

Положим по определению

$$\iota_\rho(e) = \begin{cases} \iota_{\rho_1}(e) \,\&\, \iota_{\rho'}(e), \text{ если } \rho = \rho_1\rho',\ \rho_1 \in \Re,\ \rho' \in \Re^*, \\ \mathbf{true}, \text{ если } \rho = \varepsilon; \end{cases}$$

$$\iota_\sigma(e) = \begin{cases} \iota_{\sigma'}(\iota_{\sigma_1}(e)), \text{ если } \sigma = \sigma_1\sigma',\ \sigma_1 \in \Sigma,\ \sigma' \in \Sigma^*, \\ e, \text{ если } \sigma = \varepsilon. \end{cases}$$

Здесь ι_{ρ_1} и ι_{σ_1} определяются описанием операционной среды, а $\iota_{\rho'}$ и $\iota_{\sigma'}$ — рекурсивные ссылки на соответствующие определения, приведенные выше.

Трансляционная грамматика определяет *трансляцию*:

$\tau(\mathrm{G_t}) = \{(x, [e]_\mathrm{H}) \mid \exists c_x (c_x \in \mathscr{C}(\mathrm{G_C}),\ c_x = \kappa_0 a_1 \kappa_1 a_2 \ldots \kappa_{m-1} a_m \kappa_m$ — управляющая цепочка; $\kappa_i \in (\Re \cup \Sigma)^*$ $(i = 0, 1, 2, \ldots, m)$ — цепочки контекстных символов, в которых $\rho_i \in \Re^*$ — резольверные подцепочки, $\sigma_i \in \Sigma^*$ — семантические подцепочки; $a_j \in \mathrm{T}$ $(j = 1, 2, \ldots, m)$ — терминальные символы; $x = a_1 a_2 \ldots a_m$ — входная цепочка (предложение входного языка); $e = \iota_{\sigma_m}(\ldots \iota_{\sigma_1}(e_0))$ — финальное состояние операционной среды при условии, что $\iota_{\rho_0}(e_0)$ & $\iota_{\rho_1}(e_1)$ & ... & $\iota_{\rho_m}(e_m)$ = **true**, где $e_{n+1} = \iota_{\sigma_n}(e_n)$ (n = 0, 1, ..., m); очевидно, что $e = e_{m+1}$)}. Здесь $[e]_\mathrm{H}$ обозначает проекцию точки $e \in \mathrm{E}$ на объектное подпространство H — *результат трансляции* входной цепочки x.

Другими словами, трансляция есть множество пар, в которых первая компонента есть предложение входного языка, а вторая — проекция состояния операционной среды после ее преобразований посредством семантик, входящих в соответствующую управляющую цепочку. Заметим, что управляющая цепочка

и вместе с ней предложение входного языка определяются с учетом контекстных условий, задаваемых резольверами. При этом предполагается, что резольверы вычисляются *синхронно* с семантическими преобразованиями с учетом порядка вхождения резольверных и семантических символов в управ-ляющую цепочку. Синхронизация означает, что сначала вычисляются предика-ты, ассоциированные с резольверными символами, предшествующими терми-налу a_1, в порядке следования резольверных символов. Если все они выполняются, то исполняются преобразования операционной среды, ассоциированные с семантическими символами, предшествующими терминалу a_1 (также в тек-стуальном порядке). Затем аналогичным порядком выполняются резольверы и семантики, предшествующие терминалу a_2, и т.д. Процесс заканчивается, когда успешно завершается вычисление конъюнкции предикатов, следующих за символом a_m, и исполнение соответствующих семантик.

Синтаксически правильными считаются лишь те входные цепочки, которые входят в состав управляющих, на которых выполняются все предикаты, ассоциированные с их резольверными символами. "Смысл" входного предложения x представляется семантической цепочкой $\sigma_1\sigma_2...\sigma_n$.

Если некоторому входному предложению соответствует несколько управляющих цепочек с разными семантическими составляющими, то такое предложение называется *семантически неоднозначным*. В SYNTAX-технологии семантическая неоднозначность не допускается[28]. Однако использование резольверов во многих случаях помогает избегать этой опасности.

Приведем два примера трансляционных грамматик, одна из которых трактуется как анализирующая, а другая — как порождающая. Первый пример иллюстрирует спецификацию интерпретатора арифметических выражений (калькулятора), а второй — вычисление функции факториал. Эти примеры обсуждаются далее на протяжении нескольких разделов в связи с рассмотрением различных аспектов использования SYNTAX-технологии. Как и во всех других примерах в этой книге, управляющие грамматики записываются на языке TSL[29], а операционная среда представлена на языке программирования Borland Pascal 7.0.

1.3. АНАЛИЗИРУЮЩАЯ ТРАНСЛЯЦИОННАЯ ГРАММАТИКА КАЛЬКУЛЯТОРА

Синтаксически управляемый калькулятор (спецификация которого дается в этом разделе) вычисляет арифметические выражения, операндами которых являются целые и вещественные числа, представленные в формате языка Паскаль. Допустимы арифметические операции: сложение, вычитание, умножение, деление, а также унарные плюс и минус. Унарные операции можно использовать подряд в любом количестве и последовательности. Круглые скобки также употребимы.

Построение спецификации калькулятора обсуждается в гл. 5.

---

[28] Фактически семантическая неоднозначность контролируется при построении управляющих таблиц процессоров.

[29] Описание языка TSL дано в гл. 8.

CALC — трансляционная грамматика калькулятора

MICROLEXICS

Lexical classes: d, '.', 'e', '+', '–', '*', '/', '(', ')', EOF, Escaped Symbols.
d: '0'..'9'.

Escaped Symbols: #32, #13, #10.

EOF: ''.

SYNTAX

Nonterminals: PROGRAM (ПРОГРАММА), EXPRESSION (ВЫРАЖЕНИЕ).

Terminals: 'd', '.', 'e', '+', '–', '*', '/', '(', ')', 'EOF'.

Auxiliary notions:

NUMBER (ЧИСЛО),
INTEGER NUMBER (ЦЕЛОЕ),
FRACTIONAL PART (ДРОБНАЯ_ЧАСТЬ),
EXPONENT PART (ПОРЯДОК),
DIGIT (ЦИФРА),
MONADIC OPERATION (УНАРНАЯ_ОПЕРАЦИЯ),
DYADIC OPERATION (БИНАРНАЯ_ОПЕРАЦИЯ),
OPERAND (ОПЕРАНД).

Forward pass semantics :

Reset,
Complete,
Push Mon{adic} Op{eration},
Push Dyadic Op{eration},
Push Op{eran}d,
Push Open Par{enthisis},
Unload And Discard Open Par{enthisis},
Init Int{eger Number},
App{end} Dig{it},
Set Int{eger} Part,
App{end} Exp{onent} Part,
Set Fr{actional} Part,
App{end} Fr{actional} Part,
Set Exp{onent},
Set Dig{it},
Set Sign.

PROGRAM: Reset, EXPRESSION, Complete, 'EOF'.

EXPRESSION: ((MONADIC OPERATION)*, OPERAND) # DYADIC OPERATION.

OPERAND: NUMBER, Push Op{eran}d;
Push Open Par{enthisis},'(', EXPRESSION, Unload And Discard Open Par{enthisis}, ')'.

NUMBER: INTEGER NUMBER, Set Int{eger} Part,
[FRACTIONAL PART, App{end} Fr{actional} Part],
[EXPONENT PART, App{end} Exp{onent} Part].

INTEGER NUMBER: Init Int{eger Number}, (Set Dig{it}, DIGIT, App{end} Dig{it})+.

FRACTIONAL PART: '.', INTEGER NUMBER, Set Fr{actional} Part.
EXPONENT PART: 'e', Set Sign, (['+']; '–'), INTEGER NUMBER, Set Exp{onent}.
DIGIT: 'd'.
MONADIC OPERATION: Push Mon{adic} Op{eration}, ('+' ; '–').
DYADIC OPERATION: Push Dyadic Op{eration}, ('+' ; '–' ; '*' ; '/').

ENVIRONMENT

```
const Size = 100 {Размер магазина операндов и операций};
type
  TOperation = (MonadicPlus, MonadicMinus, Plus, Minus, Mult, Division, OpenPar );
  TPrio = 0 .. 3;
  TOperationStackItem = record
                          Operation : Toperation;
                          Priority : Tprio
                        end;
 TOperandStack   = array [1 .. Size] of real {Магазин операндов};
 TOperationStack = array [1 .. Size] of TOperationStackItem {  Магазин операций };
var OperandStack : TOperandStack;
    OperationStack : TOperationStack;
    OperandStackTop, OperationStackTop : 0 .. Size;
    Number, FractionalPart : real;
    IntegerNumber, DigitsNmb, Exponent, Digit, Sign : integer;
          {Вспомогательные процедуры и функции}
function CurrentSymbol: char;
 var s : string;
 begin s := LR; CurrentSymbol := s[1] {Литерное представление лексемы} end;
function Power (A, B: integer):real {Вычисляет A^B, где A≠0, а B≥0};
  var c : real; i : integer;
  begin c := 1; for i := 1 to B do c := c*A; Power := c end ;
procedure Unload(Prio:TPrio) {Разгрузка и выполнение операций из магазина
                              операций};
  function Unloading:Boolean{Дает true, если разгрузка магазина операций
                             возможна};
  begin {Unloading} with OperationStack[OperationStackTop] do
    Unloading := (OperationStackTop>0) and (Operation<>OpenPar) and
                 (Prio<=Priority)
  end {Unloading};
  begin {Unload}
    while Unloading  do
    with OperationStack[OperationStackTop] do
```

```
    begin
      case Operation  of
          MonadicPlus:{Skip};
          MonadicMinus: OperandStack[OperandStackTop] :=
                              –OperandStack [OperandStackTop]
      else
      begin
        case Operation of
          Mult :OperandStack[OperandStackTop–1] :=
                OperandStack[OperandStackTop–1]*
                OperandStack[OperandStackTop];
          Division:OperandStack[OperandStackTop–1] :=
                   OperandStack[OperandStackTop–1]/
                   OperandStack[OperandStackTop];
          Plus: OperandStack[OperandStackTop–1] :=
                OperandStack[OperandStackTop–1]+
                OperandStack[OperandStackTop];
          Minus: OperandStack[OperandStackTop–1] :=
                  OperandStack[OperandStackTop–1]–
                  OperandStack[OperandStackTop]
        end {case};
        Dec(OperandStackTop)
      end {else-ветви}
      end {внешнего case};
      Dec (OperationStackTop)
    end {with}
  end {Unload};

        IMPLEMENTATION
              { Реализация сематик }
procedure Reset {Инициализирует вычисление входного выражения};
begin
  NewLine;
  OperationStackTop := 0      {Опустошает магазин операций};
  OperandStackTop  := 0       {Опустошает магазин операндов}
end;
procedure Complete {Завершает разгрузку магазина операций и вычисление входного
                              выражения};
begin
  Unload(0) {Разгружает магазин операций, завершая вычисление выражения};
  PrintReal(OperandStack[OperandStackTop]) {Печать результата};
  Dec(OperandStackTop){Сброс результата с вершины магазина операндов}
end;
procedure UnloadAndDiscardOpenPar {Разгрузка магазина операций до скобки и ее
                                                  сброс};
begin Unload(0); Dec(OperationStackTop) end;
```

```
procedure PushDyadicOp {Обработка бинарной операции};
var Prio : TPrio; CS : char;
begin
 CS := CurrentSymbol;
 case CS of
  '+', '–' : Prio := 1;
  '*', '/'  : Prio := 2
 end;
 Unload(Prio) {Разгружает и выполняет операции приоритета Prio и выше};
 Inc(OperationStackTop);
 with OperationStack[OperationStackTop] do
   begin
    case CS of
     '+' : Operation := Plus;
     '–' : Operation := Minus;
     '*'  : Operation := Mult;
     '/'  : Operation := Division;
    end ; Priority := Prio
  end
end ;
procedure PushMonOp {Помещает унарную операцию в магазин операций};
begin Inc(OperationStackTop);
  with OperationStack[OperationStackTop] do
  begin
    case CurrentSymbol of
     '+'  : Operation := MonadicPlus;
     '–'  : Operation := MonadicMinus
    end; Priority := 3
  end
end;
procedure PushOpd {Помещает значения операндов в магазин операндов};
begin
  Inc(OperandStackTop);
  OperandStack[OperandStackTop] := Number
end;
procedure PushOpenPar {Помещает '(' в магазин операций};
begin  Inc(OperationStackTop);
 with OperationStack[OperationStackTop] do
 begin Operation := OpenPar; Priority := 0 end
end;
procedure InitInt {Инициализация целого};
begin IntegerNumber := 0; DigitsNmb := 0 end ;
procedure SetDig {Преобразует литеру цифры в соответствующее целое};
begin Digit := LN – 1 end ;
procedure AppDig {Включение очередной цифры в целое};
begin IntegerNumber := IntegerNumber * 10 + Digit; Inc(DigitsNmb) end;
```

```
procedure SetIntPart {Инициализация целой части};
begin Number := IntegerNumber end ;
procedure SetFrPart {Установка дробной части};
var i : integer;
begin  FractionalPart := IntegerNumber;
  for i := 1 to DigitsNmb do FractionalPart := FractionalPart * 0.1
end ;
procedure AppFrPart  {Добавление дробной части к числу};
begin  Number := Number + FractionalPart end ;
procedure SetSign {Установка знака порядка};
begin if CurrentSymbol = '–'  then Sign := –1 else Sign := +1 end ;
procedure SetExp {Учет знака порядка};
begin  Exponent := IntegerNumber * Sign end ;
procedure AppExpPart {Учет порядка};
begin
  if  Exponent >= 0  then Number := Number * Power(10, Exponent)
                     else  Number := Number / Power(10, –Exponent)
end ;
```

1.4. ПОРОЖДАЮЩАЯ ТРАНСЛЯЦИОННАЯ ГРАММАТИКА, ОПЕДЕЛЯЮЩАЯ ВЫЧИСЛЕНИЕ ФУНКЦИИ FACTORIAL

В этом разделе дается пример спецификации посредством трансляционной грамматики одного вычислительного алгоритма — алгоритма вычисления функции Factorial.

Управление процессом вычисления этой функции определяется двумя правилами грамматики. Правило для начального нетерминала S описывает внешний контур управления, а правило для вспомогательного понятия F — внутренний. Исполнение конструкции S состоит из инициализации вычисления (Start) и, в зависимости от результата тестирования параметра (Valid), выполнения собственно вычисления функции, определяемого правилом для F, а также завершения вычислений (Finish) — когда предикат Valid выполняется, либо выдачи аварийного сообщения (Alarm) — когда этот предикат не выполняется.

Терминал Start подразумевает создание экземпляра объекта типа TFactorial, который инициализируется конструктором Init, доставляющим исходное значение параметра *n* и полагающим для начала поле Fac равным 1. Резольвер Valid тестирует параметр *n*, выдавая значение **true**, если он не отрицателен, и значение **false** – в противном случае. Терминал Finish уничтожает экземпляр объекта, предварительно выдавая полученный результат на экран. Терминал Alarm также уничтожает объект, но при этом сообщает о недопустимости исходного значения параметра.

Конструкция F в свою очередь циклически выполняет шаг вычислений (Step), пока текущее значение параметра *n* велико (предикат Large выполняется). Метод Step домножает текущее значение результата Fac на текущее значе-

ние параметра *n* и уменьшает его на единицу. Резольвер Large выдает значение **true**, если текущее значение параметра больше единицы, и значение **false** — в противном случае.

Предполагается, что порождающий процессор, построенный по трансляционной грамматике FACTOR, запускается из некоторой прикладной программы. Его управляющая таблица играет роль таблицы решений, которая в зависимости от сочетания различных условий относительно текущего значения параметра *n* диктует последовательность действий, ассоциированных с терминалами грамматики.

Factor — трансляционная грамматика функции *n*!
(итеративный вариант)

```
Nonterminals: S.
Terminals: 'Start', 'Finish', 'Alarm', 'Step'.
Auxiliary notions: F. {Выражение для F подставляется использующего вхождения F в
                      правило для S — открытой функции (макроподстановки)}
Forward pass resolvers: Valid, Large, Small.
                {Правила, определяющие структуру управления алгоритма вычисления n!}
S : 'Start', (Valid, (F, 'Finish'); 'Alarm').
F : (Large, 'Step')*, Small.
    ENVIRONMENT
type  PFactorial = ^TFactorial;
      TFactorial = object
                     n {Параметр}, Fac {Результат}:integer;
                     constructor Init;
                     destructor FinishDone;
                     destructor AlarmDone;
                     function Valid  : Boolean;
                     function Large : Boolean;
                     function Small : Boolean;
                     procedure Step;
                   end;
var Factorial : PFactorial; {Экземпляр объекта — создается действием Start}
                {Реализация методов}
constructor TFactorial.Init;
begin ReadLn(n); Fac := 1 end;
destructor TFactorial.FinishDone;
begin WriteLn(n, '! = ', Fac) end;
destructor TFactorial.AlarmDone;
begin WriteLn(n, ' — недопустимое значение параметра!') end;
function TFactorial.Valid : Boolean;
begin Valid := n >= 0 end;
function TFactorial.Large : Boolean;
begin Large := n > 1 end;
```

```
function TFactorial.Small : Boolean;
begin Small := Not Large end;
procedure TFactorial.Step;
begin Fac := Fac * n; Dec(n) end;

    IMPLEMENTATION
        {Реализации терминал-действий}
procedure Start; begin Factorial := New(PFactorial, Init) end;
procedure Alarm; begin Dispose(Factorial, AlarmDone) end;
procedure Step; begin  Factorial^.Step end;
        {Реализации резольверов}
function Valid: Boolean; begin Valid := Factorial^.Valid end;
function Large: Boolean; begin Large := Factorial^.Large end;
function Small: Boolean; begin Small := Factorial^.Small end;
```

Замечание. Может показаться, что резольвер Small в правиле для конструкции F не нужен. Если его убрать, то это приведет к появлению зависимости между резольверными цепочками Valid и Valid&Large[30]. С другой стороны, нелишне убедиться в том, что по завершении итерации предикат Small действиительно выполняется. Поскольку символ F обозначает вспомогательное (т.е. раскрываемое) понятие, то ясно, что в этом примере мы имеем дело с явнорегулярной трансляцией, и процессор, ее реализующий, является конечным.

1.5. АНАЛИЗИРУЮЩИЙ ПРОЦЕССОР

Процессор, реализующий трансляцию, подобно трансляционной грамматике, состоит из управляющего процессора, который строится по управляющей грамматике, и операционной среды, компилируемой по ее описанию. В разд. 1.1 неформально был описан анализирующий процессор. Здесь мы определим этот процессор и трансляцию, им реализуемую, в точных терминах.

Анализирующим процессором назовем формальную систему $\mathscr{P} = (\mathscr{P}_c, \ \mathscr{E})$, состоящую из управляющего анализирующего процессора ($\mathscr{P}_c$) и операционной среды ($\mathscr{E}$).

Управляющим анализирующим процессором назовем формальную систему $\mathscr{P}_c = (Q, \Sigma, \Gamma, \Delta, \mathfrak{R}, \delta, q_0, F)$, включающую конечное множество *состояний* (Q), *входной алфавит* (Σ), *алфавит магазинных символов* (Γ), *алфавит семантических символов* (Δ), *алфавит резольверных символов* ($\mathfrak{R}$); *управляющую таблицу* $\delta = (\delta_1, \delta_2, \delta_3)$, которая состоит из трех подтаблиц:

- *таблицы резольверов* $\delta_1 : \ Q \times (\Sigma \cup \{\varepsilon\}) \to 2^{\mathfrak{R}^*}$,
- *таблицы управляющих элементов* $\delta_2 : Q \times (\Sigma \cup \{\varepsilon\}) \times \mathfrak{R}^* \to (Q \cup \{\mathbf{Sup}\}) \times \Gamma^* \times \Delta^*$ и
- *таблицы возвратных состояний* $\delta_3 : Q \times \Gamma \times \mathfrak{R}^* \to Q$;

$q_0 \in Q$ — *начальное состояние*, $F \subseteq Q$ — *множество конечных состояний*.

[30] Чтобы убедиться в этом, достаточно произвести подстановку выражения для вспомогательного понятия F в правую часть правила для конструкции S.

Работу процессора, как принято в теории автоматов, опишем в терминах конфигураций.

Под *конфигурацией* процессора будем подразумевать совокупность (q, x, α, e), где $q \in Q$ — текущее состояние внутреннего управления процессора, $x \in \Sigma^*$ — непрочитанная часть входной цепочки, $\alpha \in \Gamma^*$ — содержимое магазина, $e \in E$ — текущее состояние операционной среды.

Начальная конфигурация — это такая, в которой $q = q_0$ (текущее состояние управления равно начальному), x — вся входная цепочка, $\alpha = \varepsilon$ (магазин пуст), $e = e_0$[31] (текущее состояние операционной среды равно начальному).

Конечная конфигурация — это такая, в которой $q \in F$ (текущее состояние управления — конечное), $x = \varepsilon$ (прочитана вся входная цепочка), $\alpha = \varepsilon$ (магазин пуст).

На множестве конфигураций введем отношение *непосредственного следования одной конфигурации после другой* следующим образом. Пусть $(q_1, ax, X\alpha, e_1)$ — текущая конфигурация. Здесь $q_1 \in Q$ — текущее состояние управления, $a \in \Sigma \cup \{\varepsilon\}$ — текущий входной символ (первый символ непрочитанной части входной цепочки), $x \in \Sigma^*$ — остаток входной цепочки, $X \in \Gamma$ — текущий верхний символ магазина, $\alpha \in \Gamma^*$ — остаток магазинной цепочки, $e_1 \in E$ — текущее состояние операционной среды.

Прежде всего определим интерпретацию резольверных и семантических цепочек. Пусть e — некоторое состояние операционной среды, $\rho \in \Re^*$ — некоторая резольверная цепочка, а $\sigma \in \Delta^*$ — некоторая семантическая цепочка. Положим по определению

$$\iota_\rho(e) = \begin{cases} \iota_{\rho_1}(e) \& \iota_{\rho_2}(e) \ldots \& \iota_{\rho_n}(e), & \text{если } \rho = \rho_1\rho_2\ldots\rho_n,\ \rho_i \in \Re,\ (i = 1, 2, \ldots, n), \\ \textbf{true}. & \text{если } n > 0. \end{cases}$$

$$\iota_\sigma(e) = \begin{cases} \iota_{\sigma_m}(\ldots\iota_{\sigma_2}(\iota_{\sigma_1}(e))\ldots), & \text{если } \sigma = \sigma_1\sigma_2\ldots\sigma_m,\ \sigma_j \in \Delta,\ (j = 1, 2, \ldots, m),\ m > 0, \\ e, & \text{если } \sigma = \varepsilon, \text{ т.е. } m = 0. \end{cases}$$

Другими словами, пустая резольверная цепочка трактуется как тождественно истинный предикат, а пустая семантическая цепочка — как пустой оператор, т.е. преобразование, не изменяющее текущего состояния операционной среды.

В текущей конфигурации в зависимости от того, что дает управляющая таблица для текущих значений входов, могут представиться следующие варианты ситуаций[32]:

---

[31] В частности, допустимо неопределенное значение e_0.

[32] Которые идентифицируются управляющим процессором в порядке их перечисления.

Случай 1 — входной символ допустим: $\delta_1(q_1, a) \neq \varnothing$. Текущий входной символ будет принят в текущем состоянии, если одна и только одна резольверная цепочка из множества $\delta_1(q_1, a)$ дает **true**. В противном случае он может быть принят в одном из возвратных состояний[33] или признан как ошибочный (в текущем контексте). Это выясняется в ходе дальнейшего анализа ситуации.

Случай 1.1 — входной символ принимается в текущем состоянии: $\exists$ единственная резольверная цепочка $\rho \in \delta_1(q_1, a) : (\iota_\rho(e_1))$. Дальнейшее движение определяется управляющим элементом $\delta_2(q_1, a, \rho)$. Пусть $\delta_2(q_1, a, \rho) = (q_2, \gamma, \sigma)$, где $q_2 \in Q$ — *переходное* состояние, $\gamma \in \Gamma^*$ — магазинная цепочка, $\sigma \in \Delta^*$ — семантическая цепочка. Процессор записывает магазинную цепочку γ над верхним символом магазина, выполняет преобразования текущего состояния операционной среды e_1, ассоциированные с семантическими символами, составляющими семантическую цепочку σ, что дает новое ее состояние $e_2 = \iota_\sigma(e_1)$, и переходит в состояние q_2. В терминах конфигураций имеем $(q_1, ax, X\alpha, e_1) \vdash (q_2, x, \gamma X\alpha, e_2)$. После этого новая конфигурация анализируется сначала.

Случай 1.2 — контекстная неоднозначность: $\exists\ \rho, \rho' \in \delta_1(q_1, a) : (\rho \neq \rho' \ \& \ \& \ \rho \neq \varepsilon \ \& \ \rho' \neq \varepsilon \ \& \ \iota_\rho(e_1) \ \& \ \iota_{\rho'}(e_1))$. Проще говоря, в этом случае в множестве $\delta_1(q_1, a)$ существует несколько разных непустых резольверных цепочек, интерпретация которых дает истину. Процессор диагностирует контекстную неоднозначность.

Случай 2 — входной символ не принимается в текущем состоянии: $\neg\exists \rho \in \delta_1(q_1, a) : (\iota_\rho(e_1))$. В этом случае либо $\delta_1(q_1, a) = \varnothing$, либо ни одна резольверная цепочка из множества $\delta_1(q_1, a)$ не дает истины. Дальнейший анализ ситуации покажет, возможно ли ε-движение или текущий входной символ ошибочен.

Случай 2.1 — ошибка: $\delta_1(q_1, \varepsilon) = \varnothing$. В этом случае ε-движение невозможно. Если значение $\delta_1(q_1, a) \neq \varnothing$, то входной символ не принимается в текущем контексте, и процессор диагностирует контекстную ошибку; если $\delta_1(q_1, a) = \varnothing$, то входной символ не подходит ни в каком контексте, и процессор диагностирует бесконтекстную ошибку.

Случай 2.2 — имеется некоторый контекстный выбор ε-движений: $\delta_1(q_1, \varepsilon) \neq \varnothing$.

Случай 2.2.1— контекстная ошибка: $\neg\exists \rho \in \delta_1(q_1, \varepsilon) : (\iota_\rho(e_1))$.

Ни одно из контекстных условий, допускающих ε-движение в текущем состоянии операционной среды, не выполняется. Процессор диагностирует контекстную ошибку.

Случай 2.2.2 — контекстная неоднозначность: $\exists \rho, \rho' \in \delta_1(q_1, \varepsilon) : (\rho \neq \rho' \ \& \ \& \ \rho \neq \varepsilon \ \& \ \rho' \neq \varepsilon \ \& \ \iota_\rho(e_1) \ \& \ \iota_{\rho'}(e_1))$. Проще говоря, в множестве $\delta_1(q_1, \varepsilon)$ существуют несколько разных непустых резольверных цепочек, интерпретация которых дает **true**. Процессор диагностирует в этом случае контекстную неоднозначность выбора ε-движения.

[33] См. разд. 2.2.

Случай 2.2.3 — однозначный выбор ε-движения: ∃ единственная резольверная цепочка $\rho \in \delta_1(q_1, \varepsilon)$, такая, что $\iota_\rho(e_1)$. Движение процессора однозначно определяется управляющим элементом $\delta_2(q_1, \varepsilon, \rho)$.

Пусть $\delta_2(q_1, \varepsilon, \rho) = (\mathbf{Sup}^{34}, \gamma, \sigma)$, где $\gamma \in \Gamma^*$, $\sigma \in \Delta^*$. Процессор выполняет переход вида $(q_1, ax, X\alpha, e_1) \vdash (q_1, ax, \gamma X\alpha, e_2)$. Здесь $e_2 = \iota_\sigma(e_1)$. Пусть магазинная цепочка, образовавшаяся после записи цепочки γ над верхним символом магазина X, есть $\gamma X\alpha = Z\beta$. Тогда фактически $(q_1, ax, \gamma X\alpha, e_2) = (q_1, ax, Z\beta, e_2)$. Если при этом $\delta_3(q_1, Z, \rho) = q_2$, то далее процессор совершает еще один шаг перехода, а именно: $(q_1, ax, Z\beta, e_2) \vdash (q_2, ax, \beta, e_2)$. Это последнее движение затрачивает верхний символ магазина, но текущий входной символ все еще остается непринятым. Новое состояние q_2 называется *возвратным*, а исходное q_1 — *подавляемым*. Далее анализ новой конфигурации начинается с начала[35]. Процессирование заканчивается благополучно, если достигается конечная конфигурация. В противном случае считается, что входная цепочка ошибочна.

Трансляцией, реализуемой процессором $\mathscr{P}$, назовем множество

$$\tau(\mathscr{P}) = \{(x, [e]_H) \mid (q_0, x, \varepsilon, e_0) \vdash^* (q, \varepsilon, \varepsilon, e),\ q \in F\}.$$

Здесь $\vdash^*$ обозначает рефлексивно-транзитивное замыкание отношения $\vdash$ на множестве конфигураций процессора. Другими словами, трансляция, реализуемая анализирующим процессором, есть некоторая функция, отображающая каждую допустимую входную цепочку в проекцию соответствующей точки пространства состояний операционной среды на объектное подпространство. Проще говоря, результат трансляции входной цепочки есть состояние той части операционной среды, которая представляет особый интерес для пользователя (например, содержание некоторого файла, картинка на экране или текст, напечатанный на принтере).

Замечание. Очевидно, что явнорегулярная трансляция реализуема конечным процессором.

1.6. РЕАЛИЗАЦИЯ КАЛЬКУЛЯТОРА ПОСРЕДСТВОМ АНАЛИЗИРУЮЩЕГО ПРОЦЕССОРА

Ранее при помощи анализирующей трансляционной грамматики CALC была специфицирована трансляция, на входе которой было арифметическое выражение над целыми и вещественными числами, представленными в формате языка Паскаль, а на выходе — вещественное число: результат вычисления входного выражения.

Здесь мы приведем анализирующий процессор (калькулятор), который построен по этой грамматике и реализует упомянутую трансляцию.

Управляющая таблица этого процессора не содержит резольверных входов (отображение δ_1 не используется), так как в грамматике CALC нет резольверов.

---

[34] Для ε-движения вместо переходного состояния всегда дается специальное значение **Sup** (**Suppress**), означающее, что следующее (возвратное) состояние надо определять по таблице δ_3.

[35] При том же текущем входном символе.

Таблица управляющих элементов (δ_2) (см. табл. 1.1) представлена в виде множества небольших табличек, каждая из которых определяет работу процессора в одном состоянии управления. Каждая табличка предваряется характеристи-кой состояния, к которому она относится. Состояние характеризуется некоторым множеством вершин управляющей граф-схемы[36], которая является графовым аналогом управляющей грамматики. Характеристика позволяет соотнести состояние процессора с некоторым множеством синтаксических позиций, достигнутых к моменту перехода в данное состояние. Вершины представлены номерами (в фигурных скобках) соответствующих записей машинного представления управляющей граф-схемы.

Все таблички имеют по четыре графы: Вход, Семантика, Магазин и Состояние. В графе Вход перечисляются допустимые в данном состоянии входные лексические классы. Например, в состоянии 1 допустимы 'd' (любая циф-ра), '+', '–' и '('. В некоторых табличках имеется также ε-вход. Он используется управляющим процессором, когда на вход поступает символ, недопустимый в текущем состоянии. В графе Семантика дается цепочка семантических символов, которые должны интерпретироваться в указанном порядке. Графа Магазин определяет цепочку символов (номеров), которые управляющий процессор должен записывать в свой магазин в указанной последовательности. Наконец, графа Состояние может содержать: (1) номер переходного состояния, в которое управляющий процессор должен перейти, (2) специальное значение **Sup**, отсылающее к таблице возвратных состояний (δ_3), и в этом случае очередное — возвратное — состояние определяется управляющим процессором по этой таблице (см. табл. 1.2), или (3) особый символ **Stop**[37], означающий конец процессирования.

Таблица 1.1

Вход	Семантика	Магазин	Состояние
1	**2**	**3**	**4**
	Состояние 1={0}		
d	Reset;InitInt;SetDig	1	2
+	Reset;PushMonOp	1	3
–	Reset;PushMonOp	1	4
(	Reset;PushOpenPar	1	5
	Состояние 2={23}		
ε	AppDig; SetIntPart; PushOpd		**Sup**
d	AppDig;SetDig		2
.	AppDig;SetIntPart		6
E	AppDig;SetIntPart		7
+	AppDig; SetIntPart; PushOpd; PushDyadicOp		8
–	AppDig; SetIntPart; PushOpd; PushDyadicOp		9
*	AppDig; SetIntPart; PushOpd; PushDyadicOp		10
/	AppDig; SetIntPart; PushOpd; PushDyadicOp		11

[36] Об управляющих граф-схемах см. разд. 1.7. Там же приведена граф-схема калькулятора, построенная по грамматике CALC примера из разд. 1.3.

[37] Этот символ встречается только в конечных состояниях.

Продолжение табл. 1.1

1	2	3	4
Состояния 3={11}; 4={13}; 8={78}; 9={81}; 10={84}; 11={86}			
d	InitInt;SetDig		2
+	PushMonOp		3
–	PushMonOp		4
(	PushOpenPar		5
Состояние 5={69}			
d	InitInt;SetDig	2	2
+	PushMonOp	2	3
–	PushMonOp	2	4
(	PushOpenPar	2	5
Состояние 6={31}			
d	InitInt;SetDig		12
Состояние 7={46}			
d	SetSign;InitInt;SetDig		13
+	SetSign		14
–	SetSign		15
Состояние 12 = {36}			
ε	AppDig;SetFrPart;AppFrPart;PushOpd		**Sup**
d	AppDig;SetDig		12
E	AppDig;SetFrPart;AppFrPart		7
+	AppDig;SetFrPart;AppFrPart;PushOpd;PushDyadicOp		8
–	AppDig;SetFrPart;AppFrPart;PushOpd;PushDyadicOp		9
*	AppDig;SetFrPart;AppFrPart;PushOpd;PushDyadicOp		10
/	AppDig;SetFrPart;AppFrPart;PushOpd;PushDyadicOp		11
Состояние 13 = {57}			
ε	AppDig;SetExp;AppExpPart;PushOpd		**Sup**
d	AppDig;SetDig		13
+	AppDig;SetExp;AppExpPart;PushOpd;PushDyadicOp		8
–	AppDig;SetExp;AppExpPart;PushOpd;PushDyadicOp		9
*	AppDig;SetExp;AppExpPart;PushOpd;PushDyadicOp		10
/	AppDig;SetExp;AppExpPart;PushOpd;PushDyadicOp		11
Состояния 14={50}; 15={52}			
d	InitInt;SetDig		13
Состояние 16={2}			
EOF	Complete		18
Состояние 17={70}			
)	UnloadAndDiscardOpenPar		19
Состояние 18={4} (конечное)			
ε			**Stop**
Состояние 19={72}			
ε			**Sup**
+	PushDyadicOp		8
–	PushDyadycOp		9
*	PushDyadycOp		10
/	PushDyadycOp		11

Заметим, что нескольким различным состояниям могут соответствовать одинаковые таблички. В таких случаях ради экономии места в табл. 1.1 одной такой табличке соотнесено несколько состояний. Например, одна табличка относится к состояниям 3, 4, 8, 9, 10 и 11. Конечные состояния маркированы признаком «конечное» (в круглых скобках) при соответствующих характеристиках (см., например, состояние 18).

Таблица возвратных состояний 1.2 представлена в компактной форме, поскольку для всех подавляемых состояний[38] (2, 12, 13 и 19) она имеет одно и то же содержание.

Таблица 1.2

Подавляемые состояния: 2, 12, 13, 19

Магазинный символ	Возвратное состояние
1	16
2	17

В первой графе перечисляются возможные магазинные символы. Когда управляющий процессор, будучи в подавляемом состоянии, снимает символ с вер-шины своего магазина, он отыскивает соответствующий вход в этой графе и получает на выходе во второй графе соответствующее возвратное состояние.

Работу процессора-калькулятора рассмотрим на примере вычисления арифметического выражения 2*(3*(1+4))+5. Она представлена в табл. 1.3. Этот процессор воспроизводит модифицированный алгоритм Дейкстры, совмещающий преобразования выражений в обратную польскую форму с одновременным их вычислением. Арифметические операции над значениями, находящимися на вершине магазина операндов, выполняются процессором в момент, когда оригинальный алгоритм пересылает их из магазина операций на выход. Результат образуется в магазине операндов (на шаге 17). После его печати магазин операндов опустошается. Вычисление заканчивается в состоянии 18, которое является конечным. При благополучном завершении вычислений магазин процессора также оказывается пустым.

Заметим, что преимущество такого синтаксически управляемого калькулятора перед обычным состоит в том, что он обнаруживает и диагностирует ошибки в записи арифметического выражения по ходу его интерпретации. Эти ошибки обнаруживаются, когда в неподавляемом состоянии поступает недопустимый в нем символ. Диагностические сообщения этого процессора приведены в разд. 6.3, где подробно рассматривается проблема обработки синтаксических ошибок.

[38] Признаком подавляемого состояния является наличие ε-входа в относящейся к нему табличке.

Протокол работы процессора-калькулятора (см. табл.1.3) состоит из двух частей, одна из которых, озаглавленная Управляющий процессор, показывает его работу, а другая, под заголовком Операционная среда описывает ее состояние в соответствующие моменты времени. Эти моменты (шаги) пронумерованы в первой графе протокола. Графа Сост. показывает текущее состояние управления процессора. В графе Вход представлены текущие входные символы, составляющие интерпретируемое арифметическое выражение. Пустая клетка в этой графе означает, что в соответствующий момент повторно анализируется входной символ, который не был принят на предыдущем шаге процессирования[39] (см., например, шаги 11, 13 и 17). В графе Магазин показано текущее состояние магазина управляющего процессора. Его вершина на каждом шаге процессирования определяется положением последней записи (для удобства читателя графа Магазин оцифрована). В графе Семантики указывается, какие семантики исполняются в соответствующий момент обработки.

В разделе Операционная среда графа Операнд представляет магазин значений операндов, а графа Операция — магазин операций, с помощью которого обеспечивается надлежащий порядок вычисления выражения в соответствии со старшинством операций и расстановкой скобок. Обе графы этих магазинов также оцифрованы, чтобы показать, сколько записей в них имеется на каждый текущий момент процессирования. Положения вершин этих магазинов определяются номерами последних записей. Остальные графы представляют текущие значения переменных, составляющих операционную среду процессора, которые используются при интерпретации данного арифметического выражения.

1.7. ПОРОЖДАЮЩИЙ ПРОЦЕССОР

Ранее уже было сделано краткое неформальное упоминание о порождаю-щем процессоре, который в отличие от анализирующего не читает входные символы, а наоборот, выводит выходные символы, точнее, трактует их как не-которые преобразования над операционной средой. Эти символы, называемые здесь *символами действий*, дублируют, по существу, семантики. Но несмотря на теоретическую избыточность возможность использования также и семантик в порождающем процессоре сохраняется из прагматических соображений: технологически семантики могут быть полезны для разукрупнения интерпретации символов действий; кроме того, они могут быть желательны из-за психологических предпочтений.

В этом разделе мы дадим точное определение порождающего процессора и трансляции, им реализуемой.

[39] Это соответствует ε-движениям, использующим символы с вершины магазина.

Таблица 1.3

УПРАВЛЯЮЩИЙ ПРОЦЕССОР							ОПЕРАЦИОННАЯ СРЕДА											
Шаг	Сост..	Вход	Магазин			Семантики	Операнд				Операция					Integer	DigNmb	Digit	Number
			1	2	3		1	2	3	4	1	2	3	4	5				
1	1	2	1			Reset;InitInt; SetDig										0	0	2	
2	2	*	1			AppDig;SetIntPart; PushOpd; PushDyadicOp	2				*					2	1	2	2
3	10	(	1			PushOpenPar	2				*	(				2	1	2	2
4	5	3	1	2		InitInt;SetDig	2				*	(				0	0	3	
5	2	*	1	2		AppDig;SetIntPart; PushOpd; PushDyadicOp	2	3			*	(	*			3	1	3	3
6	10	(	1	2		PushOpenPar	2	3			*	(	*	(		3	1	3	3
7	5	1	1	2	2	InitInt;SetDig	2	3			*	(	*	(		0	0	1	
8	2	+	1	2	2	AppDig;SetIntPart; PushOpd; PushDyadicOp	2	3	1		*	(	*	(	+	1	1	1	1
9	8	4	1	2	2	InitInt;SetDig	2	3	1		*	(	*	(	+	0	0	4	

10	2	)	1	2	2	AppDig;SetIntPart; PushOpd	2	3	1	4	*	(	*	(	+	4	1	4	4
11	17		1	2		UnloadAndDiscard	2	3	5		*	(	*			4	1	4	4
12	19	)	1	2			2	3	5		*	(	*			4	1	4	4
13	17		1			UnloadAndDiscard	2	15			*					4	1	4	4
14	19	+	1			PushDyadicOp	30				+					4	1	4	4
15	8	5	1			InitInt;SetDig	30				+					0	0	5	
16	2	EOF				AppDig;SetIntPart; PushOpd	30	5			+					5	1	5	5
17	16					Complete	35												
18	18	**Stop**																	

Управляющим порождающим процессором назовем совокупность объектов $\mathscr{P}_c = (Q, \Sigma, \Gamma, \Delta, \Re, \delta, q_0, F)$, включающую конечное множество *состояний управления* (Q), алфавит *символов действий* (Σ)[40], алфавит *магазинных символов* (Γ), алфавит *семантических символов* (Δ), алфавит *резольверных символов* ($\Re$), управляющую таблицу $\delta = (\delta_2, \delta_3)$, состоящую из двух частей[41], таблицы *управляющих элементов* $\delta_2 : Q \times \Re^* \to (Q \cup \{\mathbf{Sup}\}) \times \Gamma^* \times \Delta^* \times (\Sigma \cup \{\varepsilon\})$ и таблицы *возвратных состояний* $\delta_3 : Q \times \Gamma \times \Re^* \to Q$, а также *начального состояния* $q_0 \in Q$ и множества *конечных состояний* $F \subseteq Q$.

Операционная среда $\mathscr{E} = (E, H, \mathscr{I}_\Re, \mathscr{I}_\Delta, \mathscr{I}_\Sigma, e_0)$ включает *пространство состояний операционной среды* E (область определения предикатов и преобразований операционной среды), *объектное подпространство* H (часть операционной среды, состояние которой представляет для пользователя особый Интерес), *предикаты* $\mathscr{I}_\Re = \{\iota_\rho : E \to \{\mathbf{false}, \mathbf{true}\} \mid \rho \in \Re\}$, ассоциированные с резольверными символами, *преобразования операционной среды* $\mathscr{I}_\Delta = \{\iota_\sigma : E \to E \mid \sigma \in \Delta\}$, ассоциированные с семантическими символами, *преобразования операционной среды* $\mathscr{I}_\Sigma = \{\iota_a : E \to E \mid a \in \Sigma\}$, ассоциированные с символами действий, *начальное состояние операционной среды* $e_0 \in E$.

Работу порождающего процессора опишем в терминах конфигураций. Под *конфигурацией процессора* будем подразумевать совокупность (q, α, e), где q — текущее состояние управления, $\alpha \in \Gamma^*$ — содержимое магазина, $e \in E$ — текущее состояние операционной среды.

Начальной конфигурацией назовем такую конфигурацию, в которой состояние управления — начальное ($q = q_0$), магазин пуст ($\alpha = \varepsilon$), а состояние операционной среды равно начальному ($e = e_0$)[42].

Конечная конфигурация — это та, в которой текущее состояние управления — конечное ($q \in F$), а магазин — пуст ($\alpha = \varepsilon$).

На множестве конфигураций введем отношение *непосредственного следования одной конфигурации после другой* ($\vdash$) следующим образом. Пусть $(q_1, Z\alpha, e_1)$ — текущая конфигурация. Здесь $q_1 \in Q$ — текущее состояние управления, $Z \in \Gamma$ — текущий верхний символ магазина, $\alpha \in \Gamma^*$ — остаток магазинной цепочки, $e_1 \in E$ — текущее состояние операционной среды.

В зависимости от того, что дает управляющая таблица для текущих значений входов, могут представиться следующие варианты ситуаций, которые идентифицируются управляющим процессором в порядке их перечисления.

Случай 1 — единственное непустое движение: $\exists$ единственная резольверная цепочка $\rho \in \Re^*$: $(\iota_\rho(e_1)$ & $\delta_2(q_1, \rho) = (q_2, \beta, \sigma, a)$ & $a \in \Sigma)$. Порождающий про-

---

[40] Аналог терминалов.

[41] Отдельная таблица резольверных входов (δ_1) в порождающем процессоре не используется. Ради сохранения параллелизма с описанием анализирующего процессора, мы нумеруем части управляющей таблицы порождающего процессора так же, как в анализирующем.

[42] В частности, e_0 может быть неопределенным.

цессор в этом случае совершает следующее движение: $(q_1, Z\alpha, e_1) \vdash (q_2, \beta Z\alpha, e_2)$, где $e_2 = \iota_a(\iota_\sigma(e_1))$. Иначе говоря, если для текущего состояния q_1 существует единственная, возможно пустая, резольверная цепочка из числа возможных на втором входе таблицы δ_2, интерпретация которой дает **true**, то процессор записывает в магазин цепочку β над текущим верхним символом магазина Z, выполняет преобразования, ассоциированные с семантической цепочкой σ, затем выполняет преобразование операционной среды, ассоциированное с символом действия $a \in \Sigma$ ("порождает" символ a) и переходит в состояние q_2.

Замечание. Поскольку, как уже отмечалось ранее, идентификация вариантов ситуаций производится в порядке их перечисления, то даже если в случае 1 выполняются условия, идентифицирующие ε-движение (см. далее), предпочтение отдается непустому движению.

Случай 2 — ε-движение: $\exists$ единственная $\rho \in \Re^*$: $(\delta_2(q_1, \rho) = (\mathbf{Sup}, \beta, \sigma, \varepsilon)$ & & $\iota_\rho(e_1))$. Существует единственная резольверная цепочка ρ, допустимая в текущем состоянии, интерпретация которой дает **true** и которой соответствует управляющий элемент, определяющий ε-движение. В этом случае, порождающий процессор совершает два последовательных движения:

$$(q_1, Z\alpha, e_1) \vdash (q_1, \beta Z\alpha, e_2) = (q_1, X\gamma, e_2) \vdash (q_2, \gamma, e_2),$$

где $\beta Z\alpha = X\gamma$, $e_2 = \iota_\sigma(e_1)$, при условии, что $\delta_3(q_1, X, \rho) = q_2$. Другими словами, после записи цепочки β над верхним символом магазина (в результате чего на вершине магазина образуется символ X) выполняется интерпретация семантической цепочки σ, которая преобразует состояние операционной среды из e_1 в e_2, а затем (с затратой верхнего символа магазина X) по таблице δ_3 определяет-ся возвратное состояние управления q_2.

Естественно, ε–действие равносильно пустому оператору.

Случай 3 — контекстная неоднозначность:$\exists \rho, \rho' \in \Re^*$: $((\delta_2(q_1, \rho) = (q_2, \beta, \sigma, a)$ & & $a \in \Sigma$ & $\delta_2(q_1, \rho') = (q_2', \beta', \sigma', a')$ & $a' \in \Sigma) \vee (\delta_2(q_1, \rho) = (\mathbf{Sup}, \beta, \sigma, \varepsilon)$ & & $\delta_2(q_1, \rho') = (\mathbf{Sup}, \beta', \sigma', \varepsilon)$ & $\iota_\rho(e_1)$ & $\iota_{\rho'}(e_1)))$.

Существует несколько резольверных цепочек, которым в текущем состоянии соответствуют различные управляющие элементы, определяющие одновременно либо непустые, либо ε-движения, причем их интерпретация дает **true**. Диагностируется контекстная неоднозначность.

Случай 4 — контекстная ошибка: $\neg\exists \rho \in \Re^*$: $(\delta_2(q_1, \rho) \neq \varnothing$ & $\iota_\rho(e_1))$. Не существует ни одной резольверной цепочки, допустимой в текущем состоянии управления, интерпретация которой давала бы **true** при текущем состоянии операционной среды. Диагностируется контекстная ошибка.

Результат трансляции, реализуемой порождающим сплайновым процессором, есть проекция финального состояния операционной среды на объектное подпространство. В общем случае он зависит от ее начального состояния. Есте-

ственно определить такого рода трансляцию как множество всевозможных пар начальных состояний операционной среды и соответствующих им результатов.

Вырожденные варианты порождающего процессора. Разумеется, возможны различные вырожденные случаи, например когда не используются магазинные символы или семантики, резольверы и даже действия. Если управляющая грамматика явнорегулярна, то соответствующий управляющий процессор — конечный (нет магазина). В редких случаях, когда грамматика определяет линейную структуру управления, резольверы не используются. Отсутствие же в ней символов действий — экзотика.

Пример порождающего процессора, реализующего явнорегулярный вариант алгоритма вычисления функции Factorial, приведен далее. В этом примере семантики не используются, поскольку все необходимые преобразования операционной среды можно связать с символами действий.

Реализация функции Factorial посредством порождающего процессора. Ранее для вычисления функции $n!$ (см. разд. 1.4) была дана спецификация итеративного алгоритма. Здесь приведем управляющую таблицу соответствующего порождающего процессора, который был построен по упомянутой явнорегулярной спецификации — см. табл. 1.4. Естественно, что процессор является конечным. Поэтому его управляющая таблица фактически состоит только из одной части — таблицы управляющих элементов (δ_2). Ее графы Условие, Действие, Состояние определяют условия выбора соответствующего действия и номер переходного состояния при этом выборе. Символ Default в графе Условие обозначает выбор действия по исключению (когда все другие условия не выполняются), а пустота в этой графе обозначает тождественно истинный предикат. Реализация резольверов и действий, а также описание всей операционной среды были показаны в разд. 1.4.

Таблица 1.4

Условие	Действие	Сост.
Состояние 1		
	Start	2
Состояние 2		
Valid & Small	Finish	3
Valid & Large	Step	4
Default	Alarm	3
Состояние 3 (конечное)		
		Stop
Состояние 4		
Small	Finish	3
Large	Step	4

Замечание. Предполагается, что все условия, образующие входы для одно-го состояния, независимы[43]. Благодаря этому они могут вычисляться в любом порядке, например в том, в котором они встречаются на входе Условие в управляющей таблице. Однако Default-вход, если он существует, используется в последнюю очередь.

Рассмотрим для примера вычисление 3! В начальном состоянии 1 безусловно выполняются действия, представленные терминалом Start. Результатом является образование экземпляра объекта Factorial с полями $n=3$ и $Fac=1$. Дальнейшая обработка производится в состоянии 2. Из двух условий, определяемых таблицей для этого состояния, не считая пустого, выполняется условие Valid&Large. Поэтому исполняются действия, ассоциированные с терминалом Step. Соответствующий метод присваивает полю Fac значение $n=3$ и уменьшает n на единицу. Теперь поля объекта Factorial имеют следующие значения: $n=2$, а $Fac=3$.

Следующие действия относятся к состоянию 4. Из двух условий Small и Large выполняется последнее. Следовательно, вновь выполняются действия Step, в результате которых Fac получает значение 6, а n — значение 1.

Снова действия определяются таблицей для состояния 4. Теперь выполняется условие Small и соответственно исполняются действия Finish: экземпляр объекта Factorial уничтожается, но перед этим деструктор FinishDone печатает результат: "3!=6". Процессирование заканчивается в состоянии 3, являющемся конечным.

1.8. СПЕЦИФИКАЦИЯ ТРАНСЛЯЦИЙ ПРИ ПОМОЩИ ТРАНСЛЯЦИОННЫХ ГРАФ-СХЕМ

Трансформационный подход. Методика построения синтаксически управляемых процессоров, используемых в SYNTAX-технологии, основывается на *трансформационном подходе*: исходная трансляционная RBNF-грамматика, специфицирующая трансляцию, подвергается эквивалентным преобразованиям (в классе RBNF-грамматик), затем трансформируется в форму трансляционной граф-схемы и уже по ней строится процессор, реализующий трансляцию. Построенный процессор, в свою очередь, может подвергаться оптимизирующим эквивалентным преобразованиям. Таким образом, в технологии используется три различных формы спецификации трансляций, одна из которых (RBNF-грамматика) удобна для первоначального задания трансляций, другая (процессоры) является формой реализации трансляций, а третья (трансляционные граф-схемы) есть промежуточная форма между двумя первыми. Впрочем, она имеет и самостоятельное значение. В частности, эта форма, аналогичная синтаксическим диаграммам Вирта, может использоваться для описания синтаксиса языков и спецификации трансляций непосредственно, тем более что граф-схемы являются более гибким аппаратом для задания трансляций, чем грамматики[44]. В технологическом комплексе SYNTAX она применяется также при автоматической генерации тестов.

---

[43] При этом всегда истинное Default-условие в расчет не принимается.

[44] Действительно, существуют граф-схемы, для которых невозможно построить прообраз управляющей грамматики, преобразование которого в граф-схему давало бы оригинальную граф-схему.

Трансляционная граф-схема $\mathcal{G} = (\mathcal{G}_c, \mathcal{E})$, как и трансляционная грамматика, состоит из двух компонент: *управляющей граф-схемы* ($\mathcal{G}_c$) и *описания операционной среды* ($\mathcal{E}$).

Управляющая граф-схема является графовым аналогом управляющей грамматики. Каждое правило RBNF-грамматики представляется в виде отдельного связного ориентированного графа, вершины которого помечены терминалами или нетерминалами грамматики, а дуги — цепочками, составленными из резольверных и семантических символов, с возможными петлями, циклами и параллельными дугами (так что уместнее было бы говорить о псевдомультиграфе). Такой граф имеет две особые вершины: *начальную*, помеченную соответствующим нетерминалом с префиксом begin-, и *конечную*, помеченную тем же нетерминалом с префиксом end-. Особенность этих вершин в том, что в начальную вершину ни одна дуга не входит, а из конечной вершины ни одна дуга не выходит. Все другие — *внутренние вершины* — помечены терминальными или нетерминальными символами. В частности, дуги могут помечаться пустыми цепочками, и тогда они называются *непомеченными*. Если все дуги управляющей граф-схемы не помечены, то она называется *синтаксической граф-схемой*. Последнее, как уже отмечалось, есть не что иное, как синтаксические диаграммы Вирта.

Таким образом, в общем случае управляющая граф-схема — это несвязный помеченный граф, в котором одна компонента связности представляет одно правило управляющей грамматики. Очевидно, что управляющая граф-схема, как формальная система, вполне аналогична управляющей грамматике с той лишь разницей, что правила имеют графовую форму, т.е. $\mathcal{G}_c = (N, T, \Re, \Sigma, K, S)$, где N — нетерминалы, T — терминалы, $\Re$ — резольверы, Σ — семантики, $K = \{K_A \mid A \in N, K_A$ — компонента графа, определяющая нетерминал $A\}$, $S \in N$ — начальный нетерминал.

Компоненту K_S, определяющую начальный нетерминал S, назовем *заглавной компонентой* управляющей граф-схемы.

Что касается описания операционной среды ($\mathcal{E}$), то оно определяется точно так же, как для трансляционных грамматик.

Пусть, например, правило управляющей грамматики имеет следующий вид: $S : (r1, s1; r2, s2), 'a'^{+}$. В нем S — нетерминал, $r1$ и $r2$ — резольверы, $s1$ и $s2$ — семантики, а 'a' — терминал. Тогда это правило можно представить так, как показано на рис. 1.4.

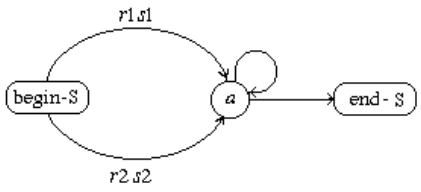


Рис. 1.4. Представление правила $S: (r1, s1; r2, s2), 'a'^{+}$.
в виде псевдомультиграфа.

Этот граф имеет параллельные дуги, помеченные контекстными цепочками *r*1*s*1 и *r*2*s*2, и петлю (не помеченную) у вершины *a*.

Как и управляющая грамматика, управляющая граф-схема порождает синтаксическое управление. Порождение управляющих цепочек реализуется с помощью *порождающих маршрутов* следующим образом. Назовем маршрут в некоторой компоненте управляющей граф-схемы *полным*, если он начинается в начальной и заканчивается в конечной вершине этой компоненты. *Следом маршрута* назовем последовательность меток вершин и дуг, составляющих этот маршрут.

Рассмотрим некоторый полный маршрут в заглавной компоненте управляющей граф-схемы и определим его след. Каждое самостоятельное вхождение нетерминала (без префиксов begin- или end-) в следе заменим на след некоторого полного маршрута в компоненте граф-схемы, определяющей этот Нетерминал. Цепочку, получаемую в результате многократного повторения таких подстановок и не содержащую ни одного самостоятельного вхождения Нетерминала, назовем *структурированной управляющей цепочкой*. Начальные и конечные метки придают ей структуру, соотносящую отдельные ее фрагменты с компонентами, их породившими.

Структурированная управляющая цепочка является аналогом скобочного представления дерева вывода в КС-грамматике. Маркеры вида begin-*A* и end-*A* играют роль структурных скобок, ограничивающих представление некоторого поддерева дерева вывода. Обычная управляющая цепочка получается из структурированной, если в ней игнорировать начальные и конечные маркеры (символы с префиксами begin- или end-).

Управляющая грамматика, представленная в приведенном примере, и эквивалентная ей граф-схема (см. рис. 1.4) порождают одно и то же синтаксическое управление, включающее, в частности, следующие структурированные управляющие цепочки:

(1) *r*1 *s*1 *a* (2) begin-S *r*2 *s*2 *a* end-S.

Сборка идентификаторов. Пусть управляющая грамматика имеет три правила:

I : Init, L, (L ; D)*, Complete.
L: Append, ('*a*' ; '*b*').
D: Append, ('0' ; '1').

Предполагается, что I, L и D — нетерминалы[45]; '*a*', '*b*', '0' и '1' — терминалы; Init, Append и Complete — семантики, которым приписывается очевидный смысл. Init инициализирует пустую строку, в которой затем собирается последовательность букв и цифр, составляющих идентификатор. Append к текущему значению этой строки присоединяет очередную литеру буквы или цифры, продолжая сборку идентификатора. Наконец, Complete завершает сборку идентификатора и передает строку в таблицу идентификаторов.

На рис. 1.5 изображена управляющая граф-схема, представляющая правила этой грамматики в виде графа из трех компонент. Нетрудно убедиться в том, что

[45] Они представляют соответственно следующие конструкции: идентификатор, буква и цифра.

управляющая грамматика и управляющая граф-схема этого примера порождают, в частности, следующую структурированную управляющую цепочку:

begin-I Init begin-L Append *a* end-L begin-L Append *b* end-L

begin-D Append 1 end-D Complete end-I.

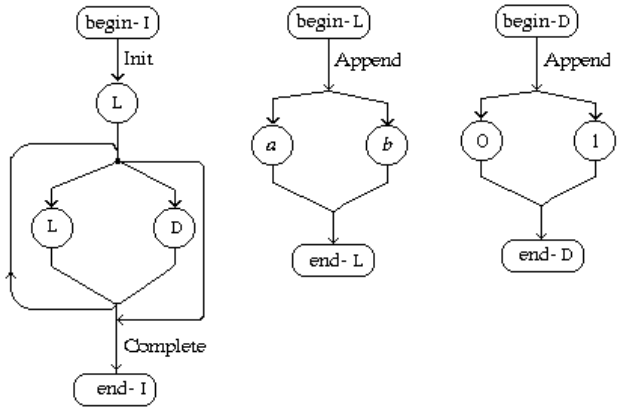


Рис. 1.5. Управляющая граф-схема, определяющая "сборку" идентификара.

Обычная управляющая цепочка, которая получается из этой структурированной путем исключения маркеров начала и конца, имеет следующий вид:

Init Append *a* Append *b* Append 1 Complete.

Постановка задачи синтаксического анализа на граф-схемах. В терминах аппарата синтаксических граф-схем задача синтаксического анализа входной цепочки формулируется как задача нахождения полного порождающего ее маршрута в данной синтаксической граф-схеме. Этот маршрут начинается в начальной вершине заглавной компоненты, проходит через некоторые другие компоненты и заканчивается в конечной вершине заглавной компоненты синтаксической граф-схемы. При этом прохождение нетерминальной вершины вызывает проход некоторого полного маршрута в компоненте, определяющей соответствующий нетерминал.

Прохождение компоненты начинается с ее начальной вершины и заканчивается в ее же конечной вершине. Приход в конечную вершину влечет возврат в нетерминальную вершину, вызвавшую обход данной компоненты. Если грамматика синтаксически неоднозначна, то порождающих маршрутов для одной входной цепочки может быть несколько. Если для данной входной цепочки не существует ни одного порождающего маршрута, то эту цепочку назовем *ошибочной*.

В применении к трансляционным граф-схемам задача синтаксического анализа может быть поставлена аналогичным образом, но с учетом контекстных условий, тестируемых предикатами. А именно: из всего множества маршрутов, порождающих данную входную цепочку, отбираются лишь те, на которых выполняются все предикаты, вычисляемые *синхронизировано* с исполнением семантик. Если в результате отбора остается несколько порождающих маршрутов, то входная цепочка *синтаксически неоднозначна* даже с учетом контекст-

ных условий. Это не опасно, если все эти маршруты семантически равнозначны. Если же в результате отбора не остается ни одного порождающего маршрута, то входная цепочка *ошибочна*.

Для определения синхронизации введем понятие *ключевой вершины маршрута*. Вершину маршрута назовем *ключевой*, если она есть начальная верши-на заглавной компоненты, терминальная или конечная вершина управляющей граф-схемы.

Синхронизация означает, что исполнение семантик и предикатов происходит в порядке следования дуг в порождающем маршруте, причем сначала вычисляется конъюнкция предикатов, помечающих участок маршрута между соседними ключевыми вершинами, а затем, если результат конъюнкции есть истина, исполняются семантики, помечающие этот же участок.

Представление управляющей RBNF-грамматики в виде управляющей граф-схемы. Как уже отмечалось, каждое правило управляющей грамматики можно представить в виде ориентированного помеченного графа. Здесь мы определим правила построения такого представления.

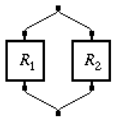
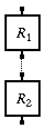
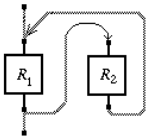
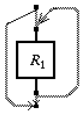
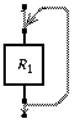
Для элементарных регулярных выражений, состоящих из одного символа, соответствующие "картинные" и линейные представления графов определяются табл. 1.5.

Таблица 1.5

№	Вид регулярного выражения	Представление в виде графа	Машинное представление графа
1	$R = \varepsilon$		α begin α+1 end
2	$R = a$ a — терминал		α begin α+1 T *a* α+*2* end
3	$R = A$ A — нетерминал		α begin α+1 N *A* α+*2* end
4	$R = \sigma$ σ — семантика		α begin α+1 FS/BS σ α+*2* end
5	$R = \rho$ ρ — резольвер		α begin α+1 FR/BR ρ α+*2* end

В конечном итоге она задает базисные элементы, к которым рекурсивно сводятся более сложные случаи, перечисленные в табл. 1.6 для регулярных формул. Рекурсия — по структуре регулярного выражения, устанавливаемой с учетом старшинства регулярных операций и расстановки скобок двух видов "(", ")" и "[", "]".

Таблица 1.6

№	Вид регулярного выражения	Представление в виде графа	Машинное представление графа		
6	$R = R_1 ; R_2$		α α+1 α+2 α+k α+k+1 α+m	begin –< /R_1/ → /R_2/ end	 α+k+1 α+m
7	$R = R_1 , R_2$		α α+1 α+k α+m	begin /R_1/ /R_2/ end	
8	$R = R_1 \# R_2$		α α+1 α+k α+k+1 α+m α+m+1	begin /R_1/ –< /R_2/ → end	 α+m+1 α+1
9	$R = R_1^*$		α α+1 α+2 α+k α+k+1	begin –< /R_1/ → end	 α+k+1 α+1
10	$R = R_1^+$		α α+1 α+k α+k+1	begin /R_1/ –< end	 α+1

Напомним, что наивысшее старшинство имеют унарные операции итерации (*) и усеченной итерации ($^+$), затем идет итерация с разделителем (*)[46], затем — конкатенация (,) и, наконец, объединение (;). Круглые скобки используются для явного указания порядка "вычисления" регулярного выражения, в то время как квадратные скобки, кроме того, еще определяют и необязательные члены порождения.

В этих таблицах T, N, FR, BR, FS и BS — типы записей, представляющих соответственно терминалы, нетерминалы, резольверы и семантики прямого и обратного просмотров[47].

Информационные поля этих записей представляют символы грамматики из соответствующих словарей. Записи типа "begin" и "end" представляют (безымянные) начальные и конечные вершины соответственно. Каждая запись имеет свой собственный номер (адрес). В таблице эти номера — относительные. Символ α обозначает фактический адрес первой записи машинного представления соответствующей регулярной формулы. Адреса используются в качестве информационных полей в записях типа –< (разветвление) и → (переход), опре-

[46] В языке TSL итерация с разделителем обозначается символом #. В дальнейшем изложении используется именно это обозначение.

[47] См. гл. 3 о челночных трансляциях.

деляющих возможный порядок обхода графа. В табл.1.6 /R_1/ и /R_2/ обозначают представления операндов регулярной формулы, которые сами являются регулярными выражениями.

Остается определить представление одного правила RBNF-грамматики в форме соответствующей компоненты графа. Пусть $A : R_A$. — правило грамматики. Чтобы получить его представление в форме графа, достаточно построить представление регулярного выражения R_A по описанным правилам и затем пометить начальную и конечную вершину нетерминальным символом A с префиксом begin- или end- соответственно.

В отличие от правил, определяющих нетерминалы, которые представляются в виде отдельных компонент управляющей граф-схемы, правила для вспомогательных понятий определяют *подставляемые* компоненты. А именно: во время генерации компоненты управляющей граф-схемы для нетерминала по правилам, определенным в табл. 1.5 и 1.6[48], каждое использующее вхождение символа вспомогательного понятия разворачивается в граф, представляющий правую часть определяющего его правила, и этот граф встраивается в генерируемую компоненту вместо той вершины, которая бы представляла этот символ, будь он нетерминалом.

Машинное (линейное) представление графа есть последовательность занумерованных записей. Такая запись состоит из двух полей, одно из которых определяет ее тип, а другое — информационное. Записи типа "begin" и "end" представляют начальную и конечную вершину соответственно. Их информационные поля содержат определяемый данной компонентой нетерминальный символ. Записи типа T и N представляют терминальные и нетерминальные вершины, а их информационные поля содержат символы соответствующих словарей. Записи типа "разветвление" (–<) и "переход" (→), в которых информационные поля содержат ссылки[49] на другие записи, определяют возможный порядок обхода компонент управляющей граф-схемы.

Далее приведем пример линейного представления управляющей граф-схемы (см. рис.1.5), построенной по грамматике, которая определяет сборку идентификаторов:

0	begin	I	11	begin	L	18	begin	D
1	FS	Init	12	FS	Append	19	FS	Append
2	N	L	13	–<	16	20	–<	23
3	–<	9	14	T	‘*a*’	21	T	‘0’
4	–<	7	15	→	17	22	→	24
5	N	D	16	T	‘*b*’	23	T	‘1’
6	→	3	17	end	L	24	end	D
7	N	D						
8	→	3						
9	FS	Complete						
10	end	I						

[48] В данных таблицах этот случай не определен.

[49] Т.е. номера других записей.

Вставки представлений вспомогательных понятий в линейном представлении граф-схем маркируются специальными записями типа "{" и "}", информационные поля которых есть символы соответствующих вспомогательных понятий. Они используются генератором диагностических сообщений для комментирования ошибок в терминах грамматики. Подробнее об этом см. разд. 6.3.

Более сложный пример управляющей граф-схемы, построенной по грамматике CALC, в форме диаграммы Вирта представлен на рис. 1.6–1.7, а ее линейное представление — на стр. 60.

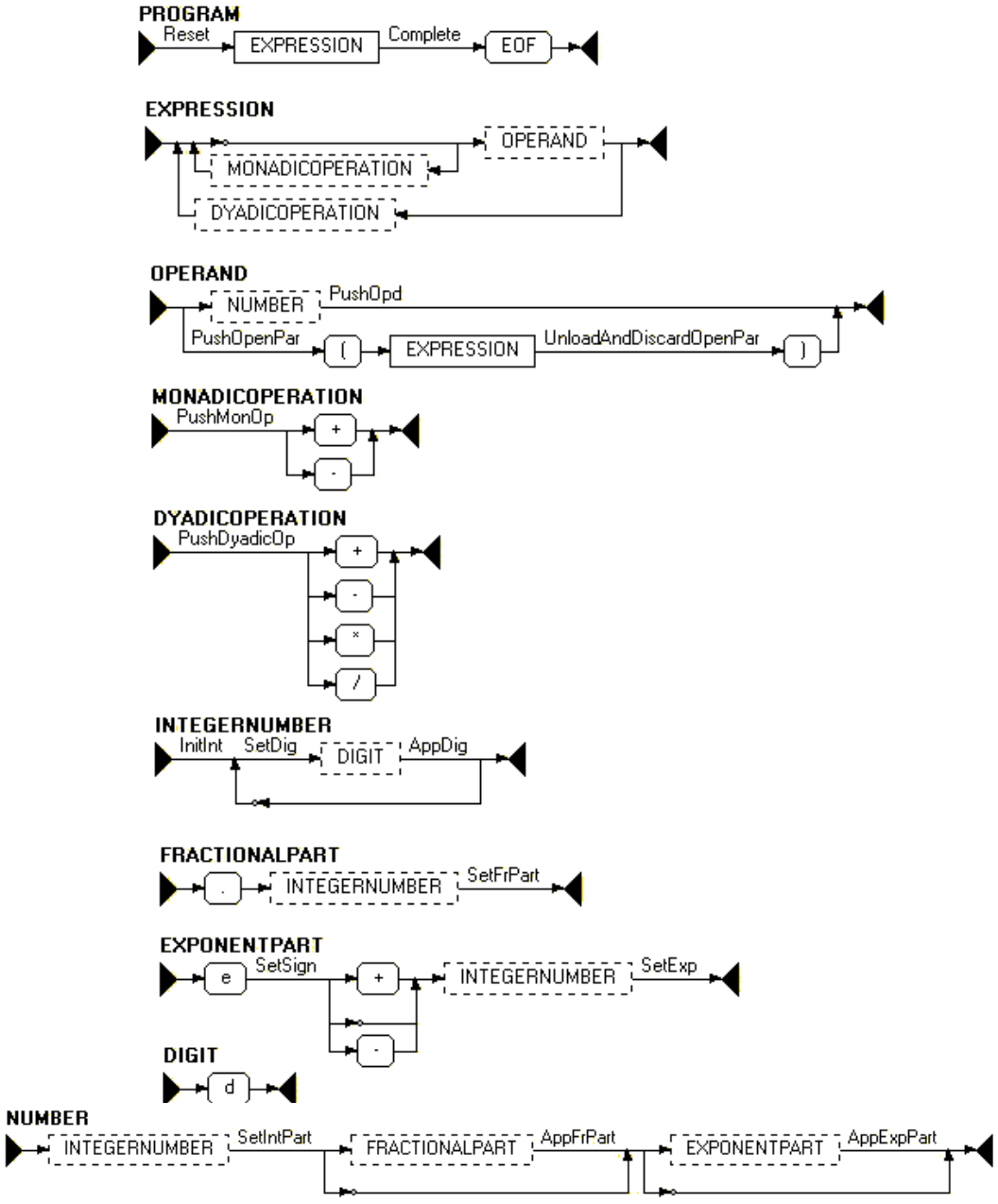


Рис. 1.6. Представление управляющей граф-схемы, построенной по грамматике CALC, в форме синтаксических диаграмм Н. Вирта.

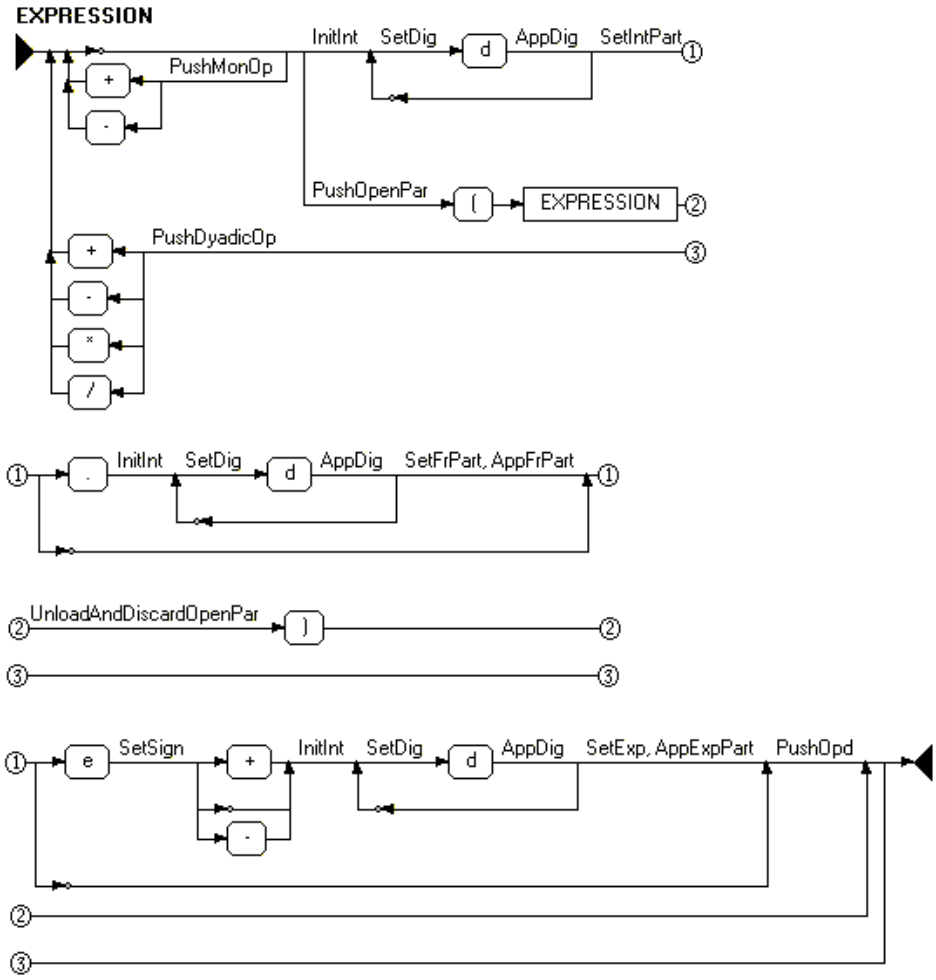


Рис. 1.7. Представление в форме синтаксической диаграммы Вирта компоненты EXPRESSION управляющей граф-схемы, построенной по грамматике CALC, после раскрытия вспомогательных понятий.

На рис. 1.6 сплошными прямоугольниками представлены использующие вхождения нетерминалов, пунктирными прямоугольниками — вспомогательные понятия, а в овалах — терминалы. На рис. 1.7 показана синтаксическая диаграмма конструкции EXPRESSION после того, как в ней раскрыты все вспомогательные понятия. Дуги помечены контекстными цепочками, определяющими трансляционную семантику соответствующей конструкции. След какого-либо маршрута от начальной вершины к конечной вершине компоненты управляющей граф-схемы, состоящий из меток терминальных вершин и меток дуг, представляет одну из управляющих цепочек, порождаемых данной компонентой. Прохождение нетерминальной вершины должно сопровождаться прохождением некоторого маршрута в соответствующей компоненте диаграммы. Отметим, что компоненты используются лишь, после того как в них раскрыты все вспомогательные понятия. Далее следует линейное представление “раскрытых” диаграмм.

0	begin	PROGRAM
1	FS	Reset
2	N	EXPRESSION
3	FS	Complete
4	T	'EOF'
5	end	PROGRAM
6	begin	EXPRESSION
7	–<	16
8	{	MONADIC OPER`ATION
9	FS	PushMonOp
10	–<	13
11	T	'+'
12	→	14
13	T	'–'
14	}	MONADIC OPERATION
15	→	7
16	{	OPERAND
17	–<	68
18	{	NUMBER
19	{	INTEGER NUMBER
20	FS	InitInt
21	FS	SetDig
22	{	DIGIT
23	T	'd'
24	}	DIGIT
25	FS	AppDig
26	–<	21
27	}	INTEGER NUMBER
28	FS	SetIntPart
29	–<	44
30	{	FRACTIONAL PART
31	T	'.'
32	{	INTEGER NUMBER
33	FS	InitInt
34	FS	SetDig
35	{	DIGIT
36	T	'd'
37	}	DIGIT
38	FS	AppDig
39	–<	34
40	}	INTEGER NUMBER
41	FS	SetFrPart
42	}	FRACTIONAL PART
43	FS	AppFrPart
44	–<	65
45	{	EXPONENT PART
46	T	'E'
47	FS	SetSign
48	–<	52
49	–<	51
50	T	'+'
51	→	53
52	T	'–'
53	{	INTEGER NUMBER
54	FS	InitInt
55	FS	SetDig
56	{	DIGIT
57	T	'd'
58	}	DIGIT
59	FS	AppDig
60	–<	55
61	}	INTEGER NUMBER
62	FS	SetExp
63	}	EXPONENT PART
64	FS	AppExpPart
65	}	NUMBER
66	FS	PushOpd
67	→	73
68	FS	PushOpenPar
69	T	'('
70	N	EXPRESSION
71	FS	UnloadDiscardOpenPar
72	T	')'
73	}	OPERAND
74	–<	89
75	{	DIADIC OPERATION
76	FS	PushDyadicOp
77	–<	80
78	T	'+'
79	→	87
80	–<	83
81	T	'–'
82	→	87
83	–<	86
84	T	'*'
85	→	87
86	T	'/'
87	}	DIADIC OPERATION
88	→	7
89	end	EXPRESSION

1.9. РЕГУЛЯРНЫЕ СПЛАЙНЫ

Методику спецификации и реализации трансляций, которая описана в предыдущих разделах этой главы, уместно назвать термином *регулярные сплайны*, поскольку она фактически основана на идее аппроксимации КС-языка регулярными множествами, аналогично тому, как в методах приближенных вычислений функции на разных участках аппроксимируются различными полиномами. Это находит свое отражение и в том, что в качестве средства задания языков используются RBNF-грамматики, правила которых определяют нетерминалы регулярными выражениями, и в том, что используемый языковой процессор на регулярных участках входной цепочки ведет себя как конечный автомат, и использует магазин лишь при переходе к другому конечному автомату и при возврате в предыдущий конечный автомат[50].

Действительно, если отвлечься от семантик и резольверов, то регулярные выражения в правых частях правил будут содержать только терминалы и нетерминалы. При этом, если, например, правило имеет вид: *A* : ... *xBy* ... . , где *A* и *B* — нетерминалы, а *x* и *y* — терминальные цепочки, то терминальное порождение фрагмента *xBy* будет иметь вид *xzy*, где *z* есть терминальное порождение *B*. Таким образом, *x* и *y* – это *регулярные фрагменты*, порождаемые собственно правилом для *A*, а *z* – фрагмент, порождаемый правилом для *B*, который тоже может состоять из регулярных фрагментов, порожденных собственно правилом для *B*, и фрагментов, порожденных другими нетерминалами. С другой стороны, нетерминалу *A* соответствует конечный автомат, который принимает (или порождает) фрагменты *x* и *y* входной цепочки *xzy*, а нетерминалу *B* соответствует конечный автомат, который принимает (или порождает) фрагменты *z*,порождаемые терминальными членами правой части правила для *B*. Соответственно, автомат *A* распознает регулярный фрагмент *x*, действуя как конечный автомат, затем происходит переключение на автомат *B* (управляющий процессор записывает символ в магазин, запоминая "точку возврата", и "передает управление" автомату *B*), который распознает какие-то регулярные фрагменты *z*, и, наконец, автомат *A* (после того как управляющий процессор извлекает символ из магазина, "возвращая управление" автомату *A*) продолжает распознавать регулярный фрагмент *y*[51].

Таким образом, уместна следующая метафора: управляющий процессор организует взаимодействие между множеством конечных автоматов, каждый из которых распознает свой регулярный фрагмент входа.

Использованию техники регулярных сплайнов в контексте объектно-синтаксического программирования посвящена следующая глава.

---

[50] Разумеется, что существуют некоторые ограничения на “гладкость” сопряжения различных конечных процессоров, реализующих данный сплайн. Они рассматриваются в разделе 5.1.

[51] Фактически взаимодействие между конечными процессорами сложнее: они могут действовать совместно, если соответствующий фрагмент входной цепочки может быть распознан несколькими семантически равнозначными конечными автоматами.

Глава 2

ОБЪЕКТНО-СИНТАКСИЧЕСКОЕ ПРОГРАММИРОВАНИЕ

2.1. ОСНОВНЫЕ ПОНЯТИЯ И ПРИМЕРЫ ОБЪЕКТНО-СИНТАКСИЧЕСКОГО ПРОГРАММИРОВАНИЯ

В гл.1 была приведена спецификация алгоритма вычисления функции Factorial в форме порождающей трансляционной грамматики и продемонстрирована ее реализация посредством порождающего сплайнового процессора. Это был первый простой пример применения техники регулярных сплайнов для спецификации и реализации вычислительных алгоритмов. Управляющая структура алгоритма вычисления функции Factorial в этом примере представлена в виде явнорегулярной грамматики[52], в которой единственный начальный нетерминал *S* определяется посредством регулярного выражения относительно терминалов (действий) и резольверов (условий). Операционная же среда фактически состоит из единственного объекта, который инкапсулирует параметр и результат функции Factorial.

Универсальный управляющий процессор, руководствуясь управляющей таблицей, сосредоточивающей в себе всю логику алгоритма вычисления этой функции, в текущем состоянии управления с учетом состояния операционной среды (текущего значения параметра функции, тестируемого посредством резольверов) вызывает соответствующее действие и определяет новое состояние управления. Выполнение действия изменяет некоторым образом состояние операционной среды. Затем в новом состоянии управления при новом состоянии операционной среды совершается очередной подобный такт процессирования.

Таким образом, терминальное порождение *S* можно рассматривать как образ некоторой программной конструкции, а экземпляр объекта Factorial — как текущее окружение, в котором эта конструкция выполняется. Точнее, под *программной конструкцией* будем подразумевать тип синтаксической структуры, ассоциированной с некоторым нетерминалом управляющей грамматики, интерпретируемой в *контексте*, который представляется некоторым типом объекта. Программная конструкция определяется *синтаксически* при помощи правила управляющей грамматики для соответствующего нетерминала (в сочетании с другими правилами, определяющими нетерминалы и вспомогательные понятия, используемые в правиле этой конструкции) и *семантически* путем описания типа объекта, представляющего *собственный контекст* этой конструкции, а также задания реализации терминал-действий и интерпретации контекстных символов.

[52] Очевидно, что в этом случае используется однофрагментный сплайн.

Экземпляр конструкции состоит из некоторой терминальной цепочки, порожденной соответствующим нетерминалом (с учетом контекста), и экземпляра объекта, представляющего *собственный контекст* данного экземпляра конструкции.

Собственный контекст конструкции — это объект, представляющий данные, к которым данная конструкция имеет непосредственный доступ через использование методов этого объекта. Эти методы обеспечивают *инициализацию и модификацию* собственного контекста данного экземпляра конструкции, а также *передачу* элементов собственного контекста конструкции в собственный контекст ее надконструкции.

Создание собственного контекста конструкции предписывается правилом управляющей грамматики при помощи включения терминал-действия 'Open' перед использующим вхождением нетерминала в правую часть правила. Это терминал-действие создает инициализированный экземпляр объекта, представляющий собственный контекст подконструкции, порождаемой этим вхождением нетерминала. Для инициализации используются данные из собственного контекста надконструкции.

Уничтожение собственного контекста конструкции предписывается правилом управляющей грамматики при помощи включения терминал-действия 'Close' после использующего вхождения нетерминала в правую часть правила. Это терминал-действие уничтожает собственный контекст конструкции, порожденной терминал-действием 'Open', предварительно передавая в контекст надконструкции необходимые данные из контекста этой конструкции.

Например, правило управляющей грамматики вида

A : ... 'Open_*B*', *B*, 'Close_*B*' ... . предписывает:

создать собственный контекст подконструкции *B* (с помощью терминал-действия 'Open_*B*', которое использует данные из контекста конструкции *A*), затем

проинтерпретировать конструкцию *B* (по правилу, определяющему Нетерминал *B*), в результате чего собственный контекст *B* получит дальнейшее развитие, и, наконец,

уничтожить собственный контекст *B* (с помощью терминал-действия 'Close_*B*') с *предварительной передачей* из контекста *B* в контекст *A* необходимых данных, которые будут использоваться конструкцией *A* в ее собственном контексте по ходу дальнейшей интерпретации этой конструкции.

Эти предписания исполняются, когда *A* становится *текущей конструкцией*.

Создание и уничтожение собственного контекста начальной конструкции, если необходимо, выполняется прикладной программой.

Для одной и той же конструкции в различных местах ее использования могут применяться различные пары терминал-действий для ее открытия и закрытия в зависимости от того, каким образом инициализируется собственный контекст для данного экземпляра конструкции или как используются данные из собственного контекста этого экземпляра конструкции ее надконструкцией.

Текущей конструкцией считается тот экземпляр конструкции, который интерпретируется в текущий момент. Он может быть идентифицирован правилом грамматики, определяющим конструкцию, и экземпляром объекта, представляющим ее собственный контекст.

Перечисленные понятия помогают не теряться в более сложной ситуации, когда вместо итеративной структуры управления используется рекурсивная. В этом случае локальные данные рекурсивной программной конструкции хранятся в своем экземпляре объекта на каждом уровне рекурсии. Например, в рекурсивной версии функции Factorial (см. далее) при каждом ее вызове образуется новый экземпляр объекта для хранения локальных данных этого вызова, причем необходимо позаботиться о связи каждого такого экземпляра с предыдущим, чтобы при завершении текущего вызова вновь обеспечить доступ к локальным данным предыдущего уровня. Это напоминает механизм управления памятью данных в стеке при вызове процедур. В рассматриваемой ООП-модели статическая цепочка локальных данных процедур образуется из экземпляров объектов, о которых идет речь. Что касается динамической цепочки, т.е. цепочки точек возврата, то она воспроизводится в магазине управляющего процессора в виде магазинных символов.

Рекурсивная реализация вычисления факториала представлена в следующем примере.

Рекурсивное вычисление функции Factorial в объектно-синтаксическом стиле. Порождающая грамматика (см. далее) определяет рекурсивную структуру управления алгоритма вычисления функции Factorial посредством двух правил.

Правило для начального нетерминала *S* задает общую организацию алгоритма: получение конкретного значения аргумента, для которого требуется вычислить значение функции Factorial; образование локального окружения конструкции *S* — первичного окружения, в котором выполняется главный экземпляр конструкции *F* (терминал-действие ‘Start’); тестирование аргумента — его значение должно быть неотрицательным (резольвер Valid); если это условие выполняется, то образуется и исполняется главный экземпляр конструкции *F* (терминал-действие ‘Open_*F*_in_*S*’ образует локальное окружение *F*, последующее вхождение *F* передает "бразды правления" правилу для *F*, затем локальное окружение *F* уничтожается терминал-действием ‘Close_*F*_in_*S*’, но результат, образованный в локальном окружении *F*, передается в первичное окружение), после чего результат, находящийся в первичном окружении, печатается, и первичное окружение уничтожается (терминал-действием ‘Finish’). Если аргумент отрицательный, то выдается сообщение об ошибке и первичное окружение уничтожается (терминал-действием ‘Alarm’).

В свою очередь, правило для *F* определяет собственно рекурсивный процесс вычисления функции. Именно, пока значение аргумента текущего экземпляра *F* велико (предикат Large выполняется), образуется и выполняется очередной экземпляр конструкции *F* с уменьшенным на единицу значением аргумента ('Open_*F*_in_*F*' , *F*, 'Close_*F*_in_*F*'). Если же у очередного экземпляра конструкции *F* значение аргумента оказывается малым (предикат Small выполняется), результат этого экземпляра конструкции полагается равным единице, его исполнение завершается и управление возвращается предыдущему экземпляру конструкции *F*. Он домножает результат конструкции, от которой он получил управление, на значение своего аргумента, и, завершая свое исполнение, передает это произведение предыдущему экземпляру конструкции *F*. В конце концов управление возвращается конструкции *S*, которая, как уже описано, выдает результат на печать и уничтожает первичное окружение.

Recfac – трансляционная грамматика *n*!
(рекурсивный вариант)

```
Nonterminals: S, F.
Terminals:'Start'          { Установить первичное окружение},
          'Finish'         { Закрыть первичное окружение},
          'Open_F_in_S'    { Установить окружение главного вызова F },
          'Close_F_in_S'   { Закрыть окружение главного вызова F },
          'Open_F_in_F'    { Установить окружение внутреннего вызова F },
          'Close_F_in_F'   { Закрыть окружение внутреннего вызова F },
          'SetOne'         { Установить результат равным 1},
          'Alarm'          { Аварийное завершение — неподходящее значение
                             параметра}.
    FORWARD PASS RESOLVERS:
          Valid            { Проверка: допустим ли параметр},
          Large            { Проверка: велик ли параметр},
          Small            { Проверка: мал ли параметр }.
          { Правила, определяющие рекурсивный алгоритм вычисления n! }
S : 'Start', ( Valid, 'Open_F_in_S', F , 'Close_F_in_S' , 'Finish' ; 'Alarm').
F :  Large , ( 'Open_F_in_F' , F , 'Close_F_in_F' ) ; Small , 'SetOne'.
    ENVIRONMENT
type  PS = ^TS;
      TS = object (TObject)
             n     { Параметр функции},
             Fac { Результат функции}: integer;
             Next : PF { Указатель на окружение главного вызова F };
             constructor Init( i : integer);
             destructor Done; virtual;
             destructor AlarmDone;
           end;
```

```
PF = ^TF;
TF = object (TS)
       Prev : Pointer;
       constructor Init ( i : integer );
       function IsLarge : Boolean;
       destructor Done; virtual;
     end;
var CurEnv: PS; { Указатель на текущее окружение }
                { Реализация методов}
constructor TS.Init ( i : integer );
begin n := i; Next := Nil end;
destructor TS.Done;
begin Writeln ( n, ' ! = ' , Fac);
       CurEnv := Nil
end;
destructor TS.AlarmDone;
begin Writeln(n, ' — неподходящий параметр') end;
constructor TF.Init ( i : integer );
begin N := i; Prev := CurEnv; Next := Nil end;
destructor TF.Done;
begin
  if n > 0 then Prev^.Fac := Fac * n else Prev^.Fac := Fac;
  Prev^.Next := Nil; CurEnv := Prev
end;
      IMPLEMENTATION
                { Реализация резольверов}
function Valid : Boolean;
begin Valid := CurEnv^.n >= 0 end;
function Large : Boolean;
begin Large := CurEnv^.n > 1 end;
function Small : Boolean;
begin Small := CurEnv^.n <= 1 end;
                { Реализация терминал-действий}
procedure Start;
var n : integer;
begin Readln(n); CurEnv := New(PS, Init(n)) end;
procedure Finish;
begin Dispose (CurEnv, Done); CurEnv := Nil end;
```

```
procedure Open_F_in_S;
begin with CurEnv^ do
  begin Next:=New(PF, Init(n)); CurEnv := Next end
end;
procedure Close_F_in_S;
begin Dispose (CurEnv, Done) end;
procedure Open_F_in_F;
begin
 with CurEnv^ do
  begin Next:=New(PF, Init (n–1)); CurEnv:=Next end
end;
procedure Close_F_in_F;
begin Dispose ( CurEnv, Done) end;
procedure Alarm;
begin Dispose (CurEnv, AlarmDone) end;
procedure SetOne;
begin CurEnv^.Fac:=1 end;
```

Управляющая граф-схема Recfac. Представление управляющей грамматики Recfac в форме управляющей граф-схемы состоит из двух компонент, каждая из которых представляет соответствующее правило:

0	begin	S	11	begin	F
1	T	‘Start’	12	–<	18
2	–<	9	13	FR	Large
3	FR	Valid	14	T	‘Open_F_in_F’
4	T	‘Open_F_in_S’	15	N	F
5	N	F	16	T	‘Close_F_in_F’
6	T	‘Close_F_in_S’	17	→	20
7	T	‘Finish’	18	FR	Small
8	→	10	19	T	‘SetOne’
9	T	‘Alarm’	20	end	F
10	end	S			

Заглавной является компонента, определяющая конструкцию *S*. Компонента, которая определяет конструкцию *F*, ссылается сама на себя, поскольку представляет рекурсивное правило. Однако благодаря тому, что этой ссылке (см. запись под номером 15) предшествует терминал-действие ‘Open_F_in_F’, недопустимая левая рекурсия места не имеет.

Управляющая таблица Recfac. Управляющая таблица порождающего сплайнового процессора, построенная по приведенной управляющей граф-схеме, состоит из двух подтаблиц — управляющих элементов (табл. 2.1) и возвратных состояний (табл. 2.2).

Таблица 2.1

Условие	Действие	Магазин	Состояние
Состояние 1 = { 0 }			
	Start		2
Состояние 2 = { 1 }			
Valid Default	Open_F_in_S Alarm		3 4
Состояние 3 = { 4 }			
Large Small	Open_F_in_F SetOne	1 1	5 6
Состояние 4 = { 9 } (конечное)			
			Stop
Состояние 5 = { 14}			
Large Small	Open_F_in_F SetOne	2 2	5 6
Состояние 6 = { 19 }			
			Sup
Состояние 7 = { 5 }			
	Close_F_in_S		9
Состояние 8 = { 15 }			
	Close_F_in_F		10
Состояние 9 = { 6 }			
	Finish		11
Состояние 10 = { 16 }			
			Sup
Состояние 11 = { 7 } (конечное)			
			Stop

Таблица 2.2

Магазинный символ	Возвратное состояние
1	7
2	8

Представленная читателю таблица возвратных состояний (табл.2.2) относится к подавляемым состояниям 6 и 10. Вход в нее в этом частном случае не зависит ни от каких условий, а только от магазинного символа.

Иллюстрация работы процессора Recfac. Рассмотрим теперь работу процессора, использующего управляющую таблицу Recfac (см. табл. 2.1 и 2.2), на примере вычисления 2!

Обработка начинается в начальном состоянии 1. Безусловно выполняются действия, ассоциированные с терминалом Start. В результате образуется экземпляр объекта типа TS, поля данных которого получают следующие значения: $n = 2$, Next = Nil (поле Fac остается неопределенным). Переменная CurEnv получает значение указателя на него. Состояние операционной среды в этот момент схематично изображено на рис.2.1. Упомянутый экземпляр объекта представлен в виде прямоугольника, помеченного символом *S*. Переменная CurEnv изображена прямоугольником со стрелкой, изображающей указатель на объект *S*. Процессор переходит в состояние 2.

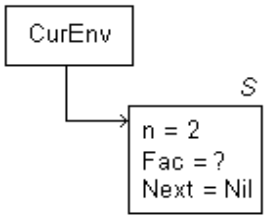


Рис.2.1. Операционная среда
после завершения действий в состоянии 1.

В состоянии 2: условие Valid выполняется, так как $n > 0$, и потому выполняются действия Open *F* in *S*. В результате образуется экземпляр объекта типа TF, представляющий локальные данные конструкции F, поля которого получают значения: $n = 2$, Next = Nil, Prev=$\uparrow S$[53](поле Fac не определено). Указатель CurEnv переводится на этот новый экземпляр объекта. Достигнутое состояние операционной среды изображено на рис.2.2. Процессор переходит в состояние 3.

[53] Очевидно, что в этом случае используется однофрагментный сплайн.

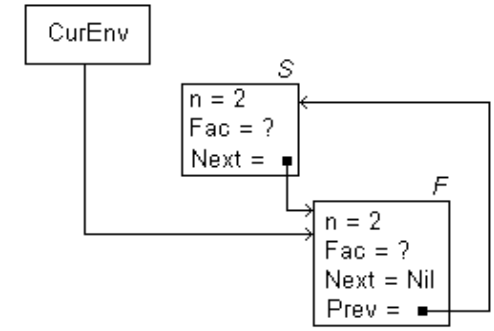


Рис.2.2. Операционная среда
после завершения действий в состоянии 2.

В состоянии 3: выполняется условие Large. Поэтому в магазин записывается 1, и выполняются действия Open_*F*_in_*F*. Образуется новый экземпляр объекта типа TF. Он изображен на рис.2.3 с меткой *F*1. Процессор переходит в состояние 5.

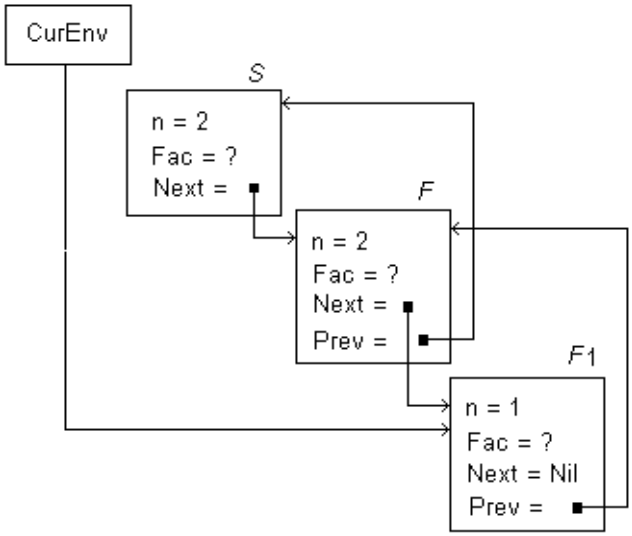


Рис.2.3. Операционная среда после завершения действий в состоянии 3.

В состоянии 5: выполняется условие Small. В магазин процессора записывается 2[54], и выполняются действия SetOne. После этого поле Fac = 1. Результат изображен на рис.2.4. Процессор переходит в состояние 6.

[54] В качестве магазинного символа.

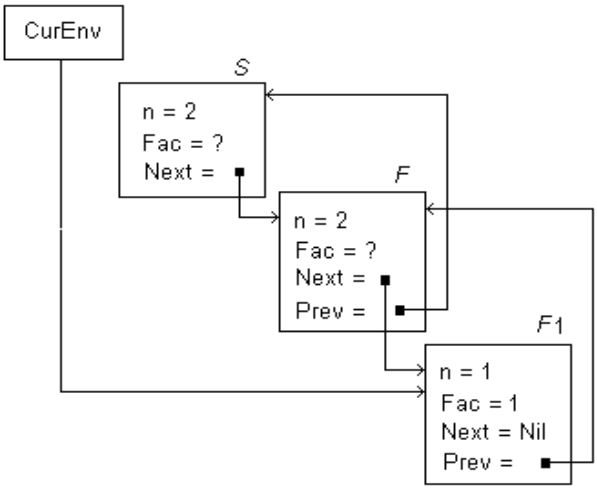


Рис. 2.4. Операционная среда после завершения действий в состоянии 5.

В состоянии 6: происходит безусловное подавление текущего состояния — по таблице возвратных состояний по текущему подавляемому состоянию (6) и верхнему символу магазина (2) определяется возвратное состояние 8. В магазине процессора остается один магазинный символ (1). Состояние операционной среды не изменяется.

В состоянии 8: безусловно выполняется действие Close *F* in *F*. Так как $n \neq 0$, то произведение $n * \text{Fac}$, равное 1, передается в предыдущее окружение *F* в его поле Fac, а экземпляр *F*1 уничтожается. Указатель CurEnv переводится на объект *F*. Состояние операционной среды в этот момент представлено на рис. 2.5. Следующее состояние управления — 10.

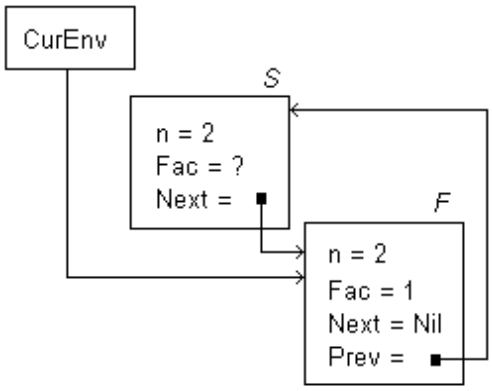


Рис. 2.5. Операционная среда
после завершения действий в состоянии 8.

В состоянии 10: текущее состояние подавляется. По таблице возвратных состояний для текущего (подавляемого) состояния (10) и верхнего символа магазина (1) определяется возвратное состояние 7. Магазин процессора опустошается. Состояние операционной среды не изменяется.

В состоянии 7: безусловно выполняется действие Close_*F*_in_*S*. Произведение $n*\text{Fac}$, равное 2, передается в поле Fac предыдущего окружения *S*, а экземпляр объекта *F* уничтожается. Указатель CurEnv переводится на объект *S*. Состояние операционной среды в этот момент изображено на рис. 2.6. Следующее состояние управления — 9.

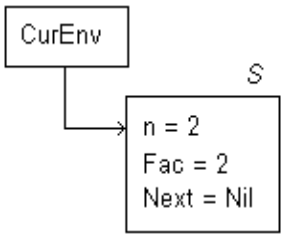


Рис. 2.6. Операционная среда
после завершения действий в состоянии 7.

В состоянии 9: безусловно выполняется операция Finish. Печатается результат: ″2! = 2″ и уничтожается последний экземпляр объекта, воплощающего локальное окружение конструкции *S* — CurEnv = Nil. Процессор переходит в состояние 11.

В состоянии 11: обработка завершается, так как это состояние конечное.

Порождающая грамматика, определяющая вычисление функции Аккермана. Без особых комментариев приведем еще один пример рекурсивного алгоритма — алгоритма вычисления функции Аккермана. По сравнению с предыдущим примером это описание более ООП-последовательное. Все используемые в нем объекты являются наследниками одного и того же абстрактного типа объекта TProto.

Определение функции Аккермана:

$\text{Ackermann}(m,n) =$ **if** $m=0 \,\&\, n>=0$ **then** $n+1$
elif $n=0 \,\&\, m>0$
then $\text{Ackermann}(m-1,1)$
elif $m>0 \,\&\, n>0$
then $\text{Ackermann}(m-1, \text{Ackermann}(m,n-1))$
fi.

Ackermann -- Трансляционная грамматика вычисления функции Аккермана

Nonterminals: MAIN, A.

Terminals: 'Start', 'Finish', 'Alarm', 'SetResult',
'Open_A_in_MAIN', 'Close_A_in_MAIN',
'Open1_A_in_A', 'Open2_A_in_A', 'Open3_A_in_A',
'Close_A_in_A'.

Forward pass resolvers: Valid, Condition1, Condition2, Condition3.

MAIN: 'Start', {Создает первичное окружение — экземпляр объекта Main. У объекта Main есть поля данных *m*, *n*, Res, mIsValid, nIsValid, причем *m* и *n* инициализируются терминал-действием 'Start'. Флажки, mIsValid и nIsValid устанавливаются предикатом Valid и используются терминал-методом 'Alarm', а Res вычисляется подконструкцией *A* конструкции MAIN.}

(Valid, { Проверяет параметры *m* и *n* на допустимость}
'Open_A_in_MAIN', { Создает экземпляр объекта *A* с передачей ему параметров *m* и *n* из Main.}
A , {Обращение к интерпретации подконструкции *A*.}
'Close_A_in_MAIN' ; { Передает результат из поля Res подконструкции *A* в поле Res конструкции MAIN и уничтожает локальный контекст подконструкции *A*.}
'Alarm', {Сообщает о недопустимости параметров *m* и *n*.}
'Finish' {Ликвидирует первичное окружение — экземпляр объекта Main, предварительно печатая вычисленное значение функции Аккермана.}
).
{Когда интерпретируется правило для конструкции *A*, ее локальный контекст (экземпляр объекта типа TA) уже создан надконструкцией MAIN или *A*. У объекта *A* есть поля данных *m*, *n* и Res, причем *m* и *n* инициализированы надконструкцией MAIN или *A*, а поле Res самой данной конструкцией *A*.}

A : Condition1, { $m = 0 \& n >= 0$ }
'SetResult' ; { $Res := n + 1$ }
Condition2, { $n = 0 \& m > 0$ }
'Open1_A_in_A', {Создание нового инициализированного экземпляра *A*}
A, {Ackermann ($m-1$, 1)}
'Close_A_in_A' ; { Передача значения поля Res подконструкции *A* в поле Res конструкции *A* и уничтожение локального контекста подконструкции *A* }
Condition3, { $m > 0 \& n > 0$ }
'Open2_A_in_A', { Создание нового инициализированного экземпляра *A* }
A, { Ackermann (m, $n-1$) }
'Close_A_in_A', { Передача результата подконструкции *A* (значения поля Res) в поле Res конструкции *A* и уничтожение локального контекста подконструкции *A* }
'Open3_A_in_A',{ Создание нового инициализированного экземпляра *A* }
A, { Ackermann ($m-1$, Ackermann (m , $n-1$)) }
'Close_A_in_A'. { Передача результата подконструкции *A* (значения поля Res) в поле Res конструкции *A* и уничтожение локального контекста подконструкции *A* }

ENVIRONMENT

```
type
    PProto = ^TProto; { Протообъект — абстрактный объект, от которого наследуются все другие объекты , представляющие локальные окружения конструкций}
    TProto = object
                Prev { Указатель на надконструкцию},
                Next {  Указатель на подконструкцию }: Pointer;
             end;
```

```
        {Локальный контекст конструкции MAIN и методы работы с ним}
    PMain = ^TMain;
    TMain = object ( TProto )
                { Вход: }      m, n,
                { Выход: }   Res : integer;
                { Флажки: } mIsValid, nIsValid : Boolean;
                { Методы: }
                constructor Init;
                destructor Done;
                destructor AlarmDone;
                procedure OpenA_in_MAIN;
                procedure Close_A_in_MAIN;
                function Valid : Boolean;
            end ;
        { Локальный контекст конструкции A и методы работы с ним .}
  PA = ^TA;
  TA = object ( TProto )
         { Параметры: }m, n,
         { Результат: }   Res : integer;
         { Методы:}
         constructor Init ( valm, valn : integer );
         destructor  Done_A_in_MAIN;
         destructor  Done_A_in_A;
         procedure SetResult;
         procerure Open1_A_in_A;
         procerure Open2_A_in_A;
         procerure Open3_A_in_A;
         procerure Close_A_in_A;
         function Condition1 : Boolean;
         function Condition2 : Boolean;
         function Condition3 : Boolean;
       end ;
var CurEnv : Pointer; { Указатель на текущее окружение — цепочку локальных кон-
                        текстов вложенных экземпляров конструкций }
  Main : PMain; { Указатель на первичное окружение}
        { Реализация методов TMain }
constructor TMain.Init;
begin
  Prev:=Nil; Next:= Nil; CurEnv:= Self;
  Writeln ('Введите параметр m:'); Readln(m);
  Writeln ('Введите параметр n:');  Readln(n)
end;
destructor TMain.Done; { Выдача результата MAIN на экран}
begin if Valid then Writeln ( 'Ackermann (', m , ',' , n , ')= ' , Res) end;
```

```
destructor TMain.AlarmDone;
begin
  if Not mIsValid then Writeln ('Значение m =', m, ' – не допустимо');
  if Not nIsValid  then Writeln ('Значение n =', n, ' – не допустимо');
  CurEnv := Nil
end;
procedure TMain.Open_A_in_MAIN;
begin Next := New (PA, Init (m , n )); CurEnv := Next end;
procedure TMain.Close_A_in_MAIN;
begin Dispose ( PA ( Next ), Done_A_in_MAIN ) end;
function TMain.Valid : Boolean;
begin
  mIsValid := m >= 0; nIsValid := n >= 0;
  Valid := mIsValid and nIsValid
end;
                { Реализация методов TA }
constructor TA.Init ( Valm, Valn : integer );
begin
  m := Valm;  n := Valn; Prev := CurEnv;
  Next := Nil; CurEnv := Self
end;
destructor TA.Done_A_in_MAIN { Выход A в MAIN };
                { Передает данные из конструкции A в надконструкцию MAIN.}
begin
  PMain(Prev)^.Res := Res;
  PProto( CurEnv)^.Next := Nil; CurEnv := Prev
end;
destructor TA.Done_A_in_A { Âûõîä èç A â A };
                { Передает данные из конструкции A в надконструкцию A. }
begin
   PA(Prev)^.Res := Res; PA ( CurEnv )^.Next := Nil; CurEnv := Prev
end;
procedure TA.SetResult;
begin  Res := n + 1 end;
procedure TA.Open1_A_in_A;
begin Next := New ( PA, Init ( m – 1, 1)) ; CurEnv := Next end;
procedure TA.Open2_A_in_A;
begin Next := New ( PA, Init ( m , n–1)); CurEnv := Next end;
procedure TA.Open3_A_in_A;
begin Next := New ( PA, Init ( m – 1, Res));  CurEnv := Next end;
```

```
procedure TA.Close_A_in_A;
begin Dispose(PA(Next), Done_A_in_A) end;
function TA.Condition1 : Boolean;
begin Condition1:=(m=0) and (n>=0) end;
function TA.Condition2 : Boolean;
begin Condition2:=(n=0) and (m>0) end;
function TA.Condition3 : Boolean;
begin Condition3:=(m>0) and (n>0) end;
```

IMPLEMENTATION

```
{ Реализация резольверов}
function  Condition1 : Boolean;
begin Condition1:=PA(CurEnv)^.Condition1 end;
function  Condition2 : Boolean;
begin Condition2:=PA(CurEnv)^.Condition2 end;
function  Condition3 : Boolean;
begin Condition3:=PA(CurEnv)^.Condition3 end;
function  Valid : Boolean;
begin Valid:=PMain(CurEnv)^.Valid end;
{ Реализация терминал-действий}
procedure Start;
begin Main:=New(PMain, Init) end;
procedure Finish;
begin Dispose(Main, Done) end;
procedure Alarm;
begin PMain(CurEnv)^.AlarmDone end;
procedure Open_A_in_MAIN;
begin PMain(CurEnv)^.Open_A_in_MAIN end;
procedure Close_A_in_MAIN;
begin PMain(CurEnv)^.Close_A_in_MAIN end;
procedure Open1_A_in_A;
begin PA(CurEnv)^.Open1_A_in_A end;
procedure Open2_A_in_A;
begin PA(CurEnv)^.Open2_A_in_A end;
procedure Open3_A_in_A;
begin PA(CurEnv)^.Open3_A_in_A end;
procedure Close_A_in_MAIN;
begin PA(CurEnv)^.Close_A_in_A  end;
```

Управляющая граф-схема Ackermann. Представление управляющей грамматики Ackermann в форме управляющей граф-схемы состоит из двух

компонент — по одной на каждое правило грамматики (MAIN — заглавная, *A* — вспомогательная рекурсивная):

```
 0  begin  MAIN
 1  T      'Start'
 2  –<     8
 3  FR     Valid
 4  T      'Open_A_in_MAIN'
 5  N      A
 6  T      'Close_A_in_MAIN'
 7  →      10
 8  T      'Alarm'
 9  T      'Finish'
10  end    MAIN
11  begin  A
12  –<     16
13  FR     Condition1
14  T      'SetResult'
15  →      29
16  –<     22
17  FR     Condition2
18  T      'Open1_A_in_A'
19  N      A
20  T      'Close_A_in_A'
21  →      29
22  FR     Condition3
23  T      'Open2_A_in_A'
24  N      A
25  T      'Close_A_in_A'
26  T      'Open3_A_in_A'
27  N      A
28  T      'Close_A_in_A'
29  end    A
```

Заметим, что компонента A трижды ссылается сама на себя, но недопустимая левосторонняя рекурсия не имеет места.

Управляющая таблица Ackermann. Управляющая таблица соответствующего порождающего процессора состоит из таблицы управляющих элементов (табл. 2.3) и таблицы возвратных состояний (табл. 2.4), относящейся к подавляемым состояниям 6, 13 и 17.

Таблица 2.3

Условие	Действие	Магазин	Состояние
1	**2**	**3**	**4**
Состояние 1 = {0}			
	Start		2
Состояние 2 = {1}			
Default	Alarm		3
Valid	Open_A_in_Main		4
Состояние 3 = {8}			
	Finish		5
Состояние 4 = {4}			
Condition1	SetResult	1	6
Condition2	Open1_A_in_A	1	7
Condition3	Open2_A_in_A	1	8

Продолжение табл. 2.3

1	2	3	4
Состояние 5 = {9} (конечное)			
			Stop
Состояние 6 = {14}			
			Sup
Состояние 7 = {18}			
Condition1	SetResult	2	6
Condition2	Open1_A_in_A	2	7
Condition3	Open2_A_in_A	2	8
Состояние 8 = {23}			
Condition1	SetResult	3	6
Condition2	Open1_A_in_A	3	7
Condition3	Open2_A_in_A	3	8
Состояние 9 = {5}			
	Close_A_in_Main		12
Состояние 10 = {19}			
	Close_A_in_A		13
Состояние 11 = {24} (конечное)			
	Close_A_in_A		14
Состояние 12 = {6} (êîíå÷íîå)			
Default			**Stop**
Состояние 13 = {20}			
			Sup
Состояние 14 = {25}			
	Open3_A_in_A		15
Состояние 15 = {26}			
Condition1	SetResult	4	6
Condition2	Open1_A_in_A	4	7
Condition3	Open2_A_in_A	4	8
Состояние 16 = {27}			
	Close_A_in_A		17
Состояние 17 = {28}			
			Sup

Таблица 2.4

Магазинный символ	Возвратное состояние
1	9
2	10
3	11
4	16

***Замечание* 1.** В табл. 2.3 отсутствует графа Семантика, которая имелась бы в том случае, если бы в грамматике использовались семантические символы. Интерпретация заданных в ней семантик должна выполняться перед исполнением операций, указанных в графе Действие.

***Замечание* 2.** Таблица возвратных состояний в этом частном случае имеет только один вход — магазинный символ, хотя в общем случае, кроме него, имелся бы вход Условие. Последний действительно использовался бы, если бы таблица управляющих элементов (табл.2.3) для какого-либо из подавляемых состояний (6, 13 или 17) выдавала значение Sup в графе Состояние в зависимости от условия. Однако это не так.

2.2. ОБЪЕКТНО-СИНТАКСИЧЕСКАЯ АРХИТЕКТУРА ПРОГРАММ

Как показывают предложенные вниманию читателя примеры, программы, написанные в стиле *объектно-синтаксического программирования*, имеют своеобразную архитектуру. Тип ее может быть выражен следующей метафорической формулой:

Программа = Объекты + Грамматика.

Это означает, что программа есть коллекция объектов, организованных для конкретного применения посредством некоторой трансляционной RBNF-грамматики. Объекты поставляют данные и методы их обработки, а управляющая грамматика определяет возможные последовательности вызовов этих методов. Грамматика порождает некоторый класс вычислений как множество цепочек терминал-действий и семантик, реализуемых посредством методов. Конкретное вычисление выбирается из этого множества цепочек в зависимости от текущего состояния операционного окружения (пространства данных).

Достоинством такой архитектуры является *инвариантность программы относительно любых преобразований ее структуры управления*. И в самом деле, реализация такой программы представляется фиксированной процедурой, работающей под управлением таблицы, которая воплощает в себе ту структуру управления, что была специфицирована исходной грамматикой. Другими словами, эта процедура эмулирует некоторый абстрактный вычислитель, а управляющая таблица играет роль программы для него. Вызывая эту фиксированную процедуру с различными таблицами в качестве ее параметров, мы настраиваем ее на определенный класс вычислений.

Синтаксический подход поддерживает новую своеобразную *парадигму программирования*. В соответствие с ней, когда вы пишете правило грамматики, то определяете некоторую программную конструкцию. При этом считается, что любая конструкция продолжает вычислительный процесс, уже выполненный составляющими ее подконструкциями, и данной конструкции требуется лишь надлежащим образом использовать результаты ее подконструкций для выработки своего собственного результата. Поэтому при написании правила грамматики, описывающего некоторую конструкцию, достаточно определить, какие действия над результатами ее подконструкций должна совершить данная конструкция. Благодаря такой *концептуальной модульности*, воплощаемой в правилах грамматики, при которой проектирование процесса обработки данных естественно структурируется иерархическим образом, разработка программы значительно облегчается.

Синтаксический подход открывает возможность *диагностировать ошибочные ситуации*[55], возникающие во время вычислений, *в терминах той предметной области, к которой относится задача*. Для этого достаточно в качестве нетерминалов или вспомогательных понятий грамматики, а также резольверных символов, использовать термины соответствующей предметной области.

2.3. ИНФОРМАЦИОННОЕ ВЗАИМОДЕЙСТВИЕ МЕЖДУ КОНСТРУКЦИЯМИ

При использовании грамматик для описания управляющей структуры программ полезно пронаблюдать аналогию между нетерминалами грамматики и замкнутыми подпрограммами, а также между вспомогательными понятиями и открытыми подпрограммами в обычном программировании.

Для каждого нетерминала существует определяющее его правило грамматики. Это правило (совместно с другими) порождает множество всевозможных конкретных реализаций соответствующей программной конструкции. В общем случае программная конструкция может состоять из других конструкций, называемых ее *подконструкциями*, которые в правиле, определяющем эту конструкцию, представляются как нетерминалы или вспомогательные понятия правой части данного правила. При этом возможны, и зачастую желательны, рекурсивные определения.

Грамматика определяет только возможные потоки управления в программе. Но операторы, входящие в этот поток и по большей части представленные (в конечном итоге) вызовами методов объектов, воплощают какие-то действия над данными, которые представляются, главным образом, полями объектов. Таким образом, с каждым нетерминалом (программной конструкцией) ассоциируется некоторый тип объекта. Его поля представляют локальные данные конструкции, а методы реализуют возможные манипуляции над этими данными.

[55] Обнаруживаемые автоматически ошибки в этом случае все являются контекстными.

Каждое вхождение нетерминала в правую часть правила порождает новый экземпляр объекта соответствующего типа — типа, ассоциированного с данной программной конструкцией. Этот экземпляр объекта является носителем локальной информации данной программной конструкции (поля данных — атрибуты конструкции) и обеспечивает средства манипуляции над этой информацией (методы). Очередной участок конкретного потока управления определяется в каждый момент процессирования в зависимости от текущего состояния операционной среды (совокупности данных) при помощи предикатов, ассоциированных с резольверными символами.

Очевидно, что должны существовать встроенные механизмы обмена информацией между программными конструкциями. Предполагаются два основных направления передачи информации: от конструкции к подконструкциям, непосредственно ее составляющим, и от подконструкции к конструкции, непосредственной составляющей которой она (подконструкция) является.

При программировании на Турбо-Паскале передачу информации от конструкции к подконструкции естественно осуществлять через фактические параметры вызова соответствующего конструктора в операторе New, образующем собственный контекст подконструкции. Фактические параметры этого вызова вычисляются в окружении конструкции, а конструктор передает эти значения в локальное окружение подконструкции. После чего подконструкция может их использовать как свои собственные локальные данные.

С другой стороны, результаты подконструкции, образуемые в полях данных объекта, представляющего ее локальные данные, можно легко сделать доступными самой конструкции, если указатель на этот объект сделать одним из полей данных объекта, представляющего локальные данные самой конструкции. Тогда, через этот указатель конструкция получит доступ к локальным данным своей подконструкции и "сама возьмет" те данные, выработанные ее подконструкцией, которые "пожелает", а затем в удобный для себя момент уничтожит объект с локальными данными подконструкции.

Естественным для Турбо-Паскаля способом передачи информации от подконструкции к конструкции является вызов деструктора объекта подконструкции в операторе Dispose, уничтожающем этот объект. Деструктор перед уничтожением объекта, представляющего локальный контекст подконструкции, осуществляет передачу тех результатов подконструкции из ее локального окружения, которые будут использованы самой конструкцией в ее локальном окружении. Но для этого деструктор должен иметь доступ к локальному окружению конструкции. Этот доступ легко обеспечить, если указатель на объект конструкции сделать одним из полей объекта подконструкции. Разумеется, оператор Dispose, о котором шла речь, должен быть "деталью" конструкции, "уничтожающей" свою подконструкцию. Поскольку подконструкция по отношению к своим подконструкциям является конструкцией, то объект с локальными данными любой конструкции должен иметь такое поле-указатель.

Итак, простой и естественный способ организации передачи информации между конструкциями состоит в том, чтобы в объекте, ассоциированном с лю-

бой конструкцией, иметь два поля связи, представляющие собой (в общем случае, нетипизированные) указатели. Один из этих указателей (Prev) — указатель на экземпляр объекта, ассоциированный с конструкцией, породившей данную конструкцию как свою подконструкцию; другой (Next) — указатель на экземпляр объекта, ассоциированного с текущей подконструкцией, которая активизирована данной конструкцией в текущий момент ее исполнения.

На рис. 2.7 представлена взаимосвязь между локальными окружениями для иерархии из трех конструкций. На нем термины "надконструкция" (*A*) и "подконструкция" (*C*) используются относительно текущей конструкции (*B*).

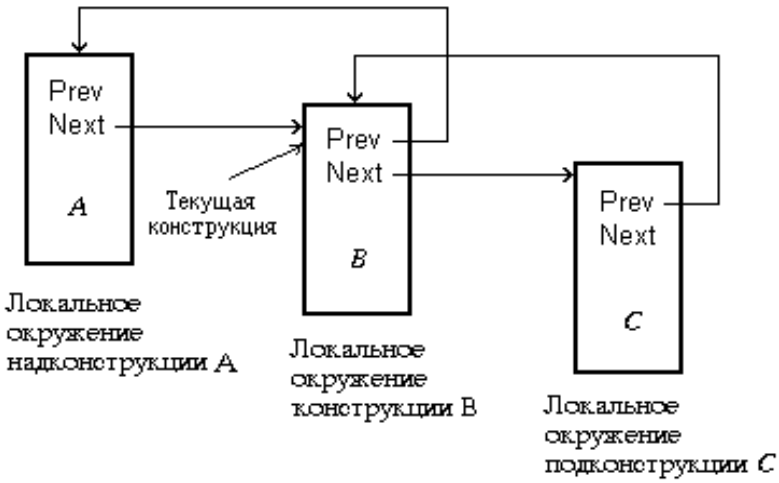


Рис. 2.7. Иерархия локальных окружений трех конструкций.

Конструкция *B* сама черпает интересующие ее данные непосредственно из полей данных экземпляра объекта, ассоциированного с текущей активной подконструкцией *C*. Именно для этой цели служит поле-указатель Next. Если у данной конструкции несколько подконструкций разного типа, то это служебное поле по необходимости представляется нетипизированным указателем. Однако в момент использования этого поля конструкция *B* всегда "знает" тип объекта, на который указывает этот указатель (и в самом деле, конструкция всегда "знает", какого типа подконструкцию она активизирует или какого типа объект создает), и в состоянии привести этот нетипизированный указатель к требуемому типу. В противном случае ничто не мешает использовать типиизированный указатель. Если конструкция простая, т.е. не содержит в своем составе других подконструкций, то это поле (указатель на объект подконструкции) равно Nil.

Когда же завершается исполнение конструкции *B*, деструктор объекта *B* передает информацию из полей данных этого объекта (или производную от нее информацию) в экземпляр объекта, ассоциированный с конструкцией *A*, используя указатель Prev в объекте *B*. При этом оператор Dispose, с которым вызывается деструктор *B*, является частью конструкции *A*. Очевидно, что у объекта *B* (как типа) может быть столько разных деструкторов, сколько разных способов использования результатов конструкции *B* определено в данной управляющей грамматике.

2.4. ОБЪЕКТНО-СИНТАКСИЧЕСКОЕ ПРОГРАММИРОВАНИЕ НА ТУРБО-ПАСКАЛЕ

Примеры вычисления функций Factorial и Ackermann показывают внешний облик объектно-синтаксических программ на Турбо-Паскале.

Такая программа состоит из "синтаксического модуля" — управляющей грамматики и "семантического модуля" — описания операционной среды, включающего описания типов объектов, ассоциированных с конструкциями, определяемыми грамматикой, и интерпретацию терминал-действий, семантик и резольверов.

Программная конструкция определяется синтаксически при помощи некоторого правила грамматики, а семантически — некоторым типом объекта. Правило включается соответственно в синтаксический модуль (раздел описания синтаксиса), а описание типа объекта, ассоциированного с конструкцией, которая описывается этим правилом, с реализацией методов этого объекта и процедур интерпретации терминал-действий и контекстных символов — в семантический модуль (раздел описания операционной среды).

Таким образом, если *A* — нетерминал, обозначающий некоторую программную конструкцию, то должно существовать правило грамматики, определяющее этот нетерминал. С другой стороны, должен быть определен тип объекта TA, представляющий локальный (т.е. собственный) контекст, создаваемый конструкцией *A*. Этот тип является прямым наследником абстрактного объекта типа TProto, имеющего только два поля данных Prev и Next типа Pointer. Таким образом, конструкция *A* будет располагать полями связи с локальными контекстами своих непосредственных над- и подконструкций.

Пусть *A* : ... *B* ... . — правило грамматики, определяющее нетерминал *A*, *B* — использующее вхождение нетерминала *B* в правой части этого правила.

Для управления контекстом конструкций использующее вхождение Нетерминала *B* должно быть заключено в рамку из терминал-действий[56], открывающих и закрывающих локальный контекст, образуемый конструкцией *A*. Соответствующий фрагмент правила для *A*, таким образом, имеет вид

A : ... 'Open_*B*' , *B*, 'Close_*B*_in_*A*' ... .

Всегда соблюдаются следующие протокольные соглашения об управлении текущим окружением:

1. Когда начинается исполнение конструкции *A*, инициированное некоторым использующим вхождением *A*, т.е. интерпретация правила для *A*, инициализированный экземпляр объекта типа TA, представляющего локальный контекст конструкции *A*, уже существует. Он создается во время интерпретации некоторого правила, использующего *A* в своей правой части. Текущее окружение в это время представляется собственным контекстом *A*.

[56] Или, если угодно, семантик.

2. Терминал-действие 'Open_*B*' исполняется уже в собственном контексте *A*. Он создает локальный контекст *B* и устанавливает его в качестве текущего. Его реализация имеет вид

```
procedure TA.Open_B;
begin Next := New ( PB, Init (...)); CurEnv := Next end;
```

Фактические параметры вызова конструктора Init (...) вычисляются в уже установленном контексте *A*. Как видно, собственный контекст подконструкции *B* создается ее конструктором, исполняемым в контексте ее надконструкции *A*. После чего текущим становится контекст, образованный конструкцией *B*.

3. В момент завершения исполнения конструкции *B* (т.е. интерпретации правила, определяющего подконструкцию *B*) текущим контекстом автоматически становится контекст вызывающей конструкции *A*. Поэтому использованием результата подконструкции *B*, зафиксированного в ее собственном контексте, и уничтожением ее локального контекста должна заниматься вызывающая конструкция *A*. Именно для этого в правиле для *A* используется терминал-действие 'Close_*B*_in_*A*'. Его реализация имеет вид

```
procedure TA.Close_B_in_A; begin Dispose ( PB(Next), Done) end;
```

Заметим, что для каждого использующего вхождения *B*, в операторе Dispose могут вызываться различные деструкторы в зависимости от способа использования данных из собственного контекста *B* в данном месте обращения к *B*.

4. Реализация деструктора объекта любой конструкции имеет вид

```
procedure TB.Done;
begin
PA (Prev)^.Res := Result; PProto (CurEnv)^.Next := Nil
end;
```

в предположении, что объект *A* имеет поле данных Res, предназначенное для сохранения результата подконструкции *B*, извлекаемого из поля данных Result объекта *B*. В общем случае, под Result можно подразумевать некоторую функцию над значениями полей данных текущего экземпляра объекта *B* (представляющего текущее окружение текущего экземпляра конструкции *B*). Результат этой функции первым оператором присваивается полю Res экземпляра объекта, представляющего собственный контекст надконструкции *B*, т.е. конструкции *A*. В это время, однако, указатель CurEnv указывает на собственный контекст конструкции *A*. Поэтому второй оператор присваивает Nil полю Next в экземпляре объекта *A* в знак того, что с этого момента никакой текущей активной подконструкции конструкции *A* не существует.

5. Если же *A*=*S*, где *S* — начальный нетерминал грамматики (заглавная программная конструкция), то ее собственный контекст должен создаваться прикладной программой.

Создание собственного контекста заглавной программной конструкции в прикладной программе можно реализовать при помощи оператора вида

```
CurEnv := New (PS, Init ( ... ));
```

а его уничтожение — посредством оператора

```
Dispose ( PS(CurEnv), Done);
```

Предполагается, что в системном семантическом модуле Environ содержатся следующие описания:

```
unit Environ;
interface
type
TProto =  object
            Prev    {Указатель на контекст надконструкции},
            Next    {Указатель на контекст подконструкции}: Pointer;
            constructor Init;
          end;
var  CurEnv : Pointer;
implementation
constructor TProto.Init;
begin Prev := CurEnv; Next := Nil end;
begin CurEnv := Nil end.
```

Переменная CurEnv используется для фиксации текущего окружения. В разделе инициализации модуля Environ ей присваивается значение Nil.

В семантическом же модуле прикладной программы (или DLL-библиотеке) должны быть описания типов объектов, ассоциированных с конструкциями, определенными в синтаксическом модуле. Этот модуль имеет вид

```
unit Application;
Interface
uses Environ;
type
PS = ^TS;
TS = object ( TProto )
        {Поля данных — локальные данные S}
      constructor Init ( ... );
      destructor Done;
    {Другие методы}
                  …
    end;
PA = ^TA;
TA = object ( TProto )
        {Поля данных — локальные данные A}
      constructor Init ( ... );
      destructor Done;
    { Другие методы }
          …
    end;  . . .
```

```
Implementation
constructor TS.Init ( ... );
begin
   Inherited Init;
{Инициализация собственного контекста S}
end;
destructor TS.Done;
begin {Передача результата S прикладной программе} end;
constructor TA.Init;
begin
   {Инициализация собственного контекста A по контексту надконструкции A}
end;
destructor TA.Done;
begin
{Передача результата A из контекста A в контекст надконструкции A}
end;
{Реализация других методов}
end.
```

Замечание. Следует иметь в виду, что все ограничения, накладываемые технологией на свойства анализирующих управляющих грамматик[57], относятся также и к порождающим грамматикам. В частности, стремление использовать семантики вместо терминал-действий в расчете на сокращение числа состояний не может быть безоглядным. Например, в спецификации функции Аккермана перевод символов 'Open1_*A*_in_*A*', 'Open2_*A*_in_*A*' è 'Open3_*A*_in_*A*' из алфавита терминалов в алфавит семантик, привел бы появлению недопустимой левосторонней рекурсии в правиле для нетерминала *A*.

2.5. ИСТОРИЧЕСКАЯ СПРАВКА

Интересно отметить, что в конце 60-х Петер Наур опубликовал статью [18], посвященную технике программирования, "в которой центральным агентом процесса управления является таблица управляющих записей. Выполнение производится программой, интерпретатором управляющих записей, которая выбирает управляющие записи и действует согласно содержанию их полей". Управляющие записи рассматриваются им как операторы некоторого языка. По его мнению так организованные программы легче поддаются систематическому тестированию в сравнении с обычными программами. Эта публикация была одним из итогов опыта, приобретенного его научным коллективом (рис. 2.8) при реализации проекта GIER ALGOL (Алгол 60).

[57] Подробнее об этом см. в гл. 3.

Рис. 2.8. Группа разработчиков компилятора GIER ALGOL (крайний справа Петер Наур, третий слева Пер Бринч Хансен — изобрататель программных мониторов).

Фактически методика объектно-синтаксического программирования, излагаемая в этой книге, является естественным развитием идеи управляющих записей, доведенным до технической реализации.

Однако, справедливости ради, следует сказать, что автор не ставил специально перед собой такой задачи. Парадигма объектно-синтаксического программирования появилась как побочный, и, можно сказать, неожиданный, результат работы над технологическим средством для автоматизации разработки компиляторов языков программирования. И уже потом вспомнилась эта давняя публикация П.Наура.

Глава 3

ЧЕЛНОЧНЫЕ ТРАНСЛЯЦИИ

3.1. РЕАЛИЗАЦИЯ ТРАНСЛЯЦИЙ ПРИ ПОМОЩИ ЧЕЛНОЧНЫХ ПРОЦЕССОРОВ

Как уже отмечалось, удобным средством для первоначальной спецификации трансляций являются трансляционные грамматики, а для реализации трансляций необходимы процессоры. Используемый в SYNTAX-технологии анализирующий процессор основывается на неявной реконструкции управляющей цепочки по предложению входного языка, которая интерпретируется по ходу ее построения. Во многих случаях такая реконструкция возможна за один просмотр входной цепочки слева направо. В более сложных случаях могут потребоваться два просмотра, первый из которых прямой, а второй — обратный.

На прямом просмотре, реализованном как анализирующий процессор, определенный в разд. 1.3, входная цепочка сканируется слева направо, и при этом происходит параллельное отслеживание множества начал всех маршрутов в управляющей граф-схеме[58], которые могли бы породить просмотренную часть входной цепочки. Такие "пучки" порождающих маршрутов фиксируются потоком состояний прямого просмотра, поскольку каждое состояние представляется как некоторое множество вершин управляющей граф-схемы, которые могли бы породить текущий входной символ или цепочку терминальных символов в качестве терминального порождения некоторой нетерминальной вершины граф-схемы.

В ходе прямого сканирования входной цепочки некоторые из маршрутов, входящие в пучок, "обрываются". Обрыв маршрута происходит тогда, когда обнаруживается, что очередной входной символ не может быть порожден на этом маршруте. Если на некотором входном символе обрываются все маршруты, то такой символ считается *ошибочным*. Если же входная цепочка не содержит ни одного ошибочного символа, то по крайней мере один полный порождающий маршрут будет пройден к концу сканирования входной цепочки. Если таких маршрутов несколько, то входная цепочка *синтаксически неоднозначна относительно прямого просмотра.* Однако это не причина для печали, так как сам факт завершения прямого просмотра является гарантией семантической однозначности входной цепочки относительно прямого просмотра, ибо достаточные для этого условия проверяются во время его построения.

[58] Напомним, что порождающие маршруты начинаются в начальной вершине заглавной компоненты граф-схемы, заканчиваются в конечной ее вершине и проходят в общем случае по нескольким компонентам граф-схемы.

Если прямой просмотр завершается успешно, то это гарантирует существование по крайней мере одного полного маршрута, порождающего просканированную входную цепочку (при условии, конечно, что контекстные условия, тестируемые резольверами обратного просмотра, выполняются). Сколько бы их ни было, все они могут быть определены путем их "попятного" прослеживания. А именно: обратный просмотр, сканируя поток состояний прямого просмотра в обратной последовательности, за концы полных маршрутов, "вытягивает" из пучка с обрывами все целые нити полных порождающих маршрутов для данной входной цепочки. Если такой маршрут единственен, то цепочка входного языка *синтаксически однозначна.* В противном случае обратный просмотр требует, чтобы все эти маршруты были семантически равнозначны относительно семантик обратного просмотра. Заметим, что параллельное продвижение по маршрутам в прямом и обратном направлениях происходит в зависимости от входных символов[59] и контекста — состояния операционной среды, тестируемого резольверами соответствующего просмотра.

Разумеется, возможность построения процессора такого рода обусловливается некоторыми ограничениями, накладываемыми на управляющую грамматику (граф-схему). Эти ограничения обсуждаются в разд. 3.3, где описывается метод построения челночного процессора. Все они следуют из априорного требования детерминированности процессора. Однако детерминированность автоматически не подразумевает синтаксическую однозначность. Допускается использование синтаксически неоднозначных управляющих грамматик (граф-схем) с тем лишь условием, что все возможные деревья вывода (или их аналоги в RBNF-грамматиках) любой синтаксически неоднозначной цепочки входного языка совпадают с точностью до меток[60] нетерминальных вершин. Очевидно, что использование резольверов существенно снижает вероятность неоднозначности.

Двухпросмотровая техника анализа, естественно, предполагает разделение семантик и резольверов по просмотрам в зависимости от того, на каком просмотре они выполняются. Это находит свое отражение в языке TSL (списки Forward/Backward semantics/resolvers).

Прямой просмотр реализуется процессором, описанным в разд. 1.3. Он управляется таблицами δ_1, δ_2, δ_3, тогда как процессор обратного просмотра управляется лишь таблицами δ_1 и δ_2 с теми особенностями, что в качестве входных символов используются состояния прямого просмотра, а магазинные цепочки — не более чем односимвольные: они представляют непосредственно возвратные состояния[61].

Обратный просмотр продолжает модификацию состояния операционной среды, начатую прямым просмотром.

В тех случаях, когда семантики обратного просмотра не используются, трансляция реализуется за один прямой просмотр. Именно такие случаи были представлены в гл. 1.

---

[59] В роли которых на обратном просмотре выступают состояния прямого просмотра.

[60] Т. е. такие деревья, если их несколько, имеют одинаковую структуру и могут отличаться разве лишь метками нетерминальных вершин.

[61] Именно поэтому обратный просмотр не использует таблицу возвратных состояний (δ_3).

Замечание. Когда требуется получить в явной форме синтаксическую структуру входного предложения, необходимы оба просмотра независимо от того, используются семантики или нет.

3.2. ЧЕЛНОЧНЫЙ СПЛАЙНОВЫЙ ПРОЦЕССОР

Дадим теперь строгое определение понятия челночного сплайнового процессора, опишем его работу и определим трансляцию, которую он реализует, в точных терминах.

Челночным сплайновым процессором назовем формальную систему $\mathscr{P}_s = (\mathscr{P}_c^f, \mathscr{P}_c^b, E)$, состоящую из управляющего сплайнового процессора прямого просмотра $\mathscr{P}_c^f$ (или просто *прямого просмотра*), управляющего сплайнового процессора обратного просмотра $\mathscr{P}_c^b$ (или просто *обратного просмотра*) и операционной среды E, компилируемой по ее описанию $\mathscr{E}$.

Прямой просмотр. Как уже отмечалось, прямой просмотр челночного процессора — это процессор, описанный в разд. 1.3. Повторим его определение в обозначениях, относящихся к челночному процессору.

Управляющим сплайновым процессором прямого просмотра назовем формальную систему $\mathscr{P}_c^f = (Q^f, \Sigma^f, \Gamma^f, \Delta^f, \Re^f, \delta^f, q_0^f, F^f)$, в которой Q^f — *множество состояний управления прямого просмотра*; Σ^f — *входной алфавит прямого просмотра*; Γ^f — *алфавит магазинных символов прямого просмотра*; Δ^f — *алфавит семантических символов прямого просмотра*; $\Re^f$ — *алфавит резольверных символов прямого просмотра*; $\delta^f = (\delta_1^f, \delta_2^f, \delta_3^f)$ — *управляющая таблица прямого просмотра*, состоящая из *таблицы резольверов прямого просмотра*: δ_1^f: $Q^f \times (\Sigma^f \cup \{\varepsilon\}) \to 2^{\Re^{f^*}}$, *таблицы управляющих элементов прямого просмотра*: δ_2^f: $Q^f \times (\Sigma^f \cup \{\varepsilon\}) \times \Re^{f^*} \to (Q^f \cup \{\mathbf{Sup}\}) \times \Gamma^{f^*} \times \Delta^{f^*}$ и *таблицы возвратных состояний*: δ_3^f: $Q^f \times \Gamma^f \times \Re^{f^*} \to Q^f$; q_0^f — *начальное состояние прямого просмотра*; $F^f \subseteq Q^f$ — *множество конечных состояний прямого просмотра*.

Обратный просмотр. *Управляющим сплайновым процессором обратного просмотра* назовем формальную систему $\mathscr{P}_c^b = (Q^b, \Sigma^b, \Gamma^b, \Delta^b, \Re^b, \delta^b, q_0^b, F^b)$, в которой Q^b — *множество состояний управления обратного просмотра*; Σ^b — *входной алфавит обратного просмотра*; Γ^b — *алфавит магазинных символов обратного просмотра*; Δ^b — *алфавит семантических символов обратного просмотра*; $\Re^b$ — *алфавит резольверных символов обратного просмотра*; $\delta^b = (\delta_1^b, \delta_2^b)$ — *управляющая таблица обратного просмотра*, состоящая из *таблицы резольверов обратного просмотра*: δ_1^b: $Q^b \times (\Sigma^b \cup \{\varepsilon\}) \to 2^{\Re^{b^*}}$ и *таблицы управляющих элементов обратного просмотра*: δ_2^b: $Q^b \times (\Sigma^b \cup \{\varepsilon\}) \times \Re^{b^*} \to (Q^b \cup \{\mathbf{Pop}\}) \times (\Gamma^b \cup \{\varepsilon\}) \times \Delta^{b^*}$; q_0^b — *начальное состояние обратного просмотра*; $F^b \subseteq Q^b$ — *множество конечных состояний обратного просмотра*.

Заметим, что управляющая таблица обратного просмотра состоит только из таблицы резольверов и таблицы управляющих элементов, тогда как таблица возвратных состояний не используется. Это связано с тем, что в случае ε-движения возвратное состояние снимается непосредственно с вершины магазина. Особенностью обратного просмотра является и то, что в магазин помещается не более одного магазинного символа одновременно. Впрочем, все это подробно рассматривается в разд. 3.3.

Операционная среда. Понятие операционной среды, включающей интерпретацию контекстных (т.е. резольверных и семантических резольверных) символов, определенное в разд. 1.2 для трансляционных грамматик и используемое в сплайновых процессорах, описанных в разд.1.3, можно естественным образом обобщить для двухпросмотрового механизма трансляции. Для этого достаточно принять во внимание, что $\Re = \Re^f \cup \Re^b$, $\Re^f \cap \Re^b = \varnothing$, $\Delta = \Delta^f \cup \Delta^b$, $\Delta^f \cap \Delta^b = \varnothing$[62] и что контекстные символы из $\Re^f$ и Δ^f интерпретируются на прямом просмотре, а резольверы и семантики из $\Re^b$ и Δ^b — на обратном. Это обобщение можно применять для трансляций, специфицируемых трансляционными грамматиками и граф-схемами, а также для трансляций, реализуемых посредством челночных сплайновых процессоров.

3.3. ФУНКЦИОНИРОВАНИЕ ЧЕЛНОЧНОГО СПЛАЙНОВОГО ПРОЦЕССОРА

Работу челночного сплайнового процессора опишем в терминах его конфигураций.

Прямой просмотр. Как уже упоминалось, обработку начинает процессор прямого просмотра, который сканирует входную цепочку, изменяя текущее состояние операционной среды. Свою работу он начинает с начальной конфигурации $(q_0^f, x, \varepsilon, e_0^f)$, где $q_0^f \in Q^f$ — начальное состояние его управления; $x \in \Sigma^{f*}$ — входная цепочка; ε означает, что первоначально магазин пуст; e_0^f — начальное состояние операционной среды.

Пусть $(q_1^f, ax, \alpha, e_1^f)$ — текущая конфигурация прямого просмотра, в которой $q_1^f \in Q^f$ — текущее состояние его управления; $ax \in \Sigma^{f*}$ — необработанная часть входной цепочки, где $a \in \Sigma^f$ — текущий входной символ, $x \in \Sigma^{f*}$ — остаток входной цепочки; $\alpha \in \Gamma^{f*}$ — текущая магазинная цепочка, причем считается, что на вершине магазина находится крайний левый символ цепочки α; e_1^f — текущее состояние операционной среды. Очередное движение определяется управляющей таблицей прямого просмотра в точности, как описано в разд. 1.3, если в этом описании положить $Q = Q^f$, $\Sigma = \Sigma^f$, $\Gamma = \Gamma^f$, $\Delta = \Delta^f$, $\Re = \Re^f$, $\delta = \delta^f$ (соответственно $\delta_1 = \delta_1^f$, $\delta_2 = \delta_2^f$, $\delta_3 = \delta_3^f$), $q_0 = q_0^f$, $F = F^f$.

Прямой просмотр заканчивает свою работу тогда, когда он достигает одного из своих конечных состояний при пустом магазине.

---

[62] В трансляционных грамматиках для обозначения семантических символов вместо символа Δ используется символ Σ, причем предполагается, что $\Sigma = \Sigma^f \cup \Sigma^b$ и $\Sigma^f \cap \Sigma^b = \varnothing$.

Вся последовательность движений прямого просмотра может быть представлена следующим образом:

$$(q_0^{\mathrm{f}}, x, \varepsilon, e_0^{\mathrm{f}}) \overset{*}{\underset{\mathscr{P}_{\mathrm{c}}^{\mathrm{f}}}{\vdash}} (p^{\mathrm{f}}, \varepsilon, \varepsilon, e^{\mathrm{f}}),$$

здесь $p^{\mathrm{f}} \in \mathrm{F}^{\mathrm{f}}$. Достигнутое состояние операционной среды e^{f} является начальным для обратного просмотра. Подразумевается, что вся последовательность состояний управления прямого просмотра регистрируется на его выходе.

Обратный просмотр, сканируя в обратном порядке последовательность состояний управления прямого просмотра, продолжает изменение состояний операционной среды, начиная со своей начальной конфигурации, которая имеет вид $(q_0^{\mathrm{b}}, p^{\mathrm{f}} \ldots\ q_0^{\mathrm{f}}, \varepsilon,\ e^{\mathrm{f}})$.

Пусть $(q_1^{\mathrm{b}},\ q_{\mathrm{k}}^{\mathrm{f}}\ q_{\mathrm{k}-1}^{\mathrm{f}}\ \ldots\ q_0^{\mathrm{f}}, \alpha,\ e_1^{\mathrm{b}})$ — текущая конфигурация обратного просмотра. Очередное движение обратного просмотра определяется его управляющей таблицей в зависимости от ситуации.

Случай 1 — входной символ допустим: $\delta_1^{\mathrm{b}}(q_1^{\mathrm{b}},\ q_{\mathrm{k}}^{\mathrm{f}}) \neq \varnothing$. Текущий входной "символ" $q_{\mathrm{k}}^{\mathrm{f}}$ будет принят в текущем состоянии, если интерпретация одной и только одной резольверной цепочки из множества $\delta_1^{\mathrm{b}}(q_1^{\mathrm{b}},\ q_{\mathrm{k}}^{\mathrm{f}})$ дает **true**. В случае, если ни одна из них не дает **true**, входной символ может быть принят в одном из возвратных состояний[63] или признан как ошибочный в текущем контексте[64]. Если несколько резольверных цепочек дает **true**, то ситуация контекстно неоднозначна[65].

Управляющий процессор разбирает возможные варианты ситуаций в перечисленном ниже порядке.

Случай 1.1 — входной символ принимается в текущем состоянии: $\exists$ единственная резольверная цепочка $r \in \delta_1^{\mathrm{b}}(q_1^{\mathrm{b}},\ q_k^{\mathrm{f}}) : \iota_r(e_1)$. Движение определяется управляющим элементом $\delta_2^{\mathrm{b}}(q_1^{\mathrm{b}},\ q_k^{\mathrm{f}},\ r)$. Пусть $\delta_2^{\mathrm{b}}(q_1^{\mathrm{b}},\ q_k^{\mathrm{f}},\ r) = (q_2^{\mathrm{b}}, \gamma, \sigma)$, где $q_2^{\mathrm{b}} \in \mathrm{Q}^{\mathrm{b}}$ — *переходное состояние*, $\gamma \in (\Gamma^{\mathrm{b}} \cup \{\varepsilon\})$ — магазинная цепочка[66], $\sigma \in \Delta^{\mathrm{b}^*}$ — семантическая цепочка. Процессор записывает магазинную цепочку γ над верхним символом магазина[67], выполняет преобразования текущего состояния операционной среды e_1^{b}, ассоциированные с семантическими символами из цепочки σ, что дает новое ее состояние: $e_2^{\mathrm{b}} = \iota_\sigma(e_1^{\mathrm{b}})$, и переходит в состояние q_2^{b}. Текущим входным символом становится следующий символ на входе. В терминах конфигураций имеем:

$$(q_1^{\mathrm{b}}, q_k^{\mathrm{f}}\, q_{k-1}^{\mathrm{f}} \ldots\ q_0^{\mathrm{f}}, \alpha, e_1^{\mathrm{b}}) \underset{\mathscr{P}_{\mathrm{c}}^{\mathrm{b}}}{\vdash} (q_2^{\mathrm{b}}, q_{k-1}^{\mathrm{f}} \ldots\ q_0^{\mathrm{f}}, \gamma\alpha, e_2^{\mathrm{b}}).$$

После этого новая конфигурация анализируется с начала.

---

[63] См. случай 2.2.3.

[64] См. случай 2.

[65] См. случай 1.2.

[66] Одно-символьная или пустая.

[67] Если цепочка пустая, то в магазин ничего не пишется, иначе ее единственный символ записывается над верхним символом магазина.

Случай 1.2 — контекстная неоднозначность: $\exists r, r' \in \delta^{b}{}_{1}(q_1{}^{b}, q_k{}^{f})$: $(r \neq r' \;\&\; r \neq \varepsilon \;\&\; r' \neq \varepsilon \;\&\; \iota_r(e_1{}^{b}) \;\&\; \iota_{r'}(e_1{}^{b}))$. Проще говоря, в этом случае в множестве $\delta^{b}_{1}(q_1{}^{b}, q_k{}^{f})$ существует несколько разных непустых резольверных цепочек, интерпретация которых дает **true**. Обратный просмотр диагностирует контекстную неоднозначность.

Случай 2 — входной символ не принимается в текущем состоянии. Здесь либо $\delta^{b}_{1}(q_1{}^{b}, q_k{}^{f}) = \varnothing$, либо ни одна резольверная цепочка из множества $\delta^{b}_{1}(q_1{}^{b}, q_k{}^{f})$ не дает **true**. Дальнейший анализ ситуации покажет, возможно ли ε-движение или текущий входной символ ошибочен.

Случай 2.1 — контекстная ошибка: $\delta^{b}_{1}(q_1{}^{b}, \varepsilon) = \varnothing$. Здесь ε-движение невозможно. Текущий входной символ не принимается. Обратный просмотр диагностирует контекстную ошибку.

Случай 2.2 — имеется некоторый контекстный выбор ε-движений: $\delta^{b}_{1}(q_1{}^{b}, \varepsilon) \neq \varnothing$.

Случай 2.2.1 — контекстная ошибка: $\neg \exists r \in \delta^{b}_{1}(q_1{}^{b}, \varepsilon) : (\iota_r(e_1{}^{b}))$. Ни одно из контекстных условий, допускающих ε-движение в текущем состоянии операционной среды, не выполняется. В этом случае процессор диагностирует контекстную ошибку.

Случай 2.2.2 — контекстная неоднозначность ε-движения: $\exists\, r, r' \in \delta^{b}_{1}(q_1{}^{b}, \varepsilon)$: $(r \neq r' \;\&\; r \neq \varepsilon \;\&\; r' \neq \varepsilon \;\&\; \iota_r(e_1{}^{b}) \;\&\; \iota_{r'}(e_1{}^{b}))$. Проще говоря, в множестве $\delta^{b}_{1}(q_1{}^{b}, \varepsilon)$ существуют несколько разных непустых резольверных цепочек, интерпретация которых при текущем состоянии операционной среды дает истину. Обратный просмотр диагностирует контекстную неоднозначность выбора ε-движения.

Случай 2.2.3 — однозначный выбор ε-движения: $\exists$ единственная $r \in \delta^{b}_{1}(q_1{}^{b}, \varepsilon)$: $(\iota_r(e_1{}^{b}))$. Дальнейшее движение процессора однозначно определяется управляющим элементом $\delta^{b}_{2}(q_1{}^{b}, \varepsilon, r)$.

Пусть $\delta^{b}_{2}(q_1{}^{b}, \varepsilon, r) = (\mathbf{Pop}^{68}, \gamma, \sigma)$, где $\gamma \in (\Gamma^{b} \cup \{\varepsilon\})$, $\sigma \in \Delta^{b*}$. В этом случае обратный просмотр выполняет переход вида

$$(q_1{}^{b}, q_k{}^{f}\, q^{f}_{k-1} \ldots\, q_0{}^{f}\, , \alpha, e_1{}^{b}) \vdash_{\mathscr{P}_c{}^{b}} (q_1{}^{b}, q_k{}^{f}\, q^{f}_{k-1} \ldots\, q_0{}^{f}, \gamma\alpha, e_2{}^{b}).$$

Здесь $e_2{}^{b} = \iota_\sigma(e_1{}^{b})$.

Пусть магазинная цепочка, образовавшаяся после записи γ над верхним символом магазина, есть $\gamma\alpha = X\beta$. Тогда имеем

$$(q_1{}^{b}, q_k{}^{f}\, q^{f}_{k-1} \ldots\, q_0{}^{f}, \gamma\alpha, e_2{}^{b}) = (q_1{}^{b}, q_k{}^{f}\, q^{f}_{k-1} \ldots\, q_0{}^{f}, X\beta, e_2{}^{b}).$$

Далее процессор совершает еще один шаг перехода, а именно:

$$(q_1{}^{b}, q_k{}^{f}\, q^{f}_{k-1} \ldots\, q_0{}^{f}, X\beta, e_2{}^{b}) \vdash_{\mathscr{P}_c{}^{b}} (q_2{}^{b}, q_k{}^{f}\, q^{f}_{k-1} \ldots\, q_0{}^{f}\, , \beta, e_2{}^{b}).$$

---

[68] Для ε-движения вместо переходного состояния всегда дается специальное значение **Pop**: следующее (возвратное) состояние следует брать с вершины магазина.

Здесь $q_2^b = X$. Это последнее движение затрачивает верхний символ магазина, но текущий входной символ все еще остается не принятым. Новое состояние q_2^b называется *возвратным*, а исходное q_1^b — *подавляемым*. Далее анализ новой конфигурации начинается по той же схеме при том же текущем входном символе.

Процессирование заканчивается благополучно, если достигается конечная конфигурация, характеризуемая тем, что достигнуто одно из конечных состояний управления обратного просмотра, прочитана вся последовательность состояний прямого просмотра и магазин пуст. В противном случае считается, что цепочка на входе челночного процессора ошибочна. Вся последовательность движений обратного просмотра может быть представлена следующим образом:

$$(q_0^b, p^f \ldots q_0^f, \varepsilon, e^f) \underset{\mathscr{P}_c^b}{\overset{*}{\vdash}} (p^b, \varepsilon, \varepsilon, e^b).$$

Здесь $p^b \in F^b$ — конечное состояние обратного просмотра, прочитана вся последовательность состояний прямого просмотра и магазин пуст.

Результатом трансляции цепочки *x*, прочитанной на входе челночного процессора, считается состояние "интересной" части операционной среды, т. е. $[e^b]_H$.

Итак, *трансляцией, реализуемой челночным сплайновым процессором,* которую мы также будем называть *челночной трансляцией*, называется множество пар:

$$\tau(\mathscr{P}_s) = \{(x, [e^b]_H) \mid (q_0^f, x, \varepsilon, e_0^f) \underset{\mathscr{P}_c^f}{\overset{*}{\vdash}} (p^f, \varepsilon, \varepsilon, e^f), p^f \in F^f,$$
$$(q_0^b, p^f \ldots q_0^f, \varepsilon, e^f) \underset{\mathscr{P}_c^b}{\overset{*}{\vdash}} (p^b, \varepsilon, \varepsilon, e^b), p^b \in F^b\}.$$

***Замечание* 1.** Напомним, что результат интерпретации пустой резольверной цепочки всегда **true**. Поэтому, если таблица резольверных символов прямого или обратного просмотра для любого входа, включая и ε-вход, дает множество резольверных цепочек, содержащее пустую цепочку, то она используется для входа в таблицу управляющих элементов только в том случае, когда интерпретация всех непустых резольверных цепочек этого множества дает **false**. При этом истину может доставить не более чем одна непустая цепочка из этого множества.

***Замечание* 2.** Очевидно, что явнорегулярная челночная трансляция реализуема конечным челночным процессором. Это значит, что оба его просмотра — конечные.

3.4. ПОСТРОЕНИЕ ЧЕЛНОЧНЫХ СПЛАЙНОВЫХ ПРОЦЕССОРОВ

Пусть задана трансляционная граф-схема $\mathscr{G} = (\mathscr{G}_c, \mathscr{E})$, которая, как мы знаем, состоит из управляющей граф-схемы ($\mathscr{G}_c$) и описания операционной среды ($\mathscr{E}$). Пусть $\mathscr{G}_c = (N, T, \Re, \Sigma, K, S)$, где N — алфавит нетерминалов; T — алфавит терминалов; $\Re = \Re^f \cup \Re^b (\Re^f \cap \Re^b = \varnothing)$ — алфавит резольверных символов, состоящий из резольверов прямого ($\Re^f$) и обратного ($\Re^b$) просмот-

ров; $\Sigma = \Sigma^f \cup \Sigma^b$ ($\Sigma^f \cap \Sigma^b = \varnothing$) — алфавит семантических символов, состоящий из семантик прямого (Σ^f) и обратного (Σ^b) просмотров; K — множество компонент управляющей граф-схемы, каждая из которых определяет некоторый нетерминал; S — начальный нетерминал.

Опишем метод построения челночного сплайнового процессора $\mathscr{P}_s = (\mathscr{P}_c^f, \mathscr{P}_c^b, E)$, где $\mathscr{P}_c^f, \mathscr{P}_c^b$ — управляющие процессоры соответственно прямого и обратного просмотров; E — операционная среда. Поскольку операционная среда компилируется по ее описанию, заданному в составе трансляционной граф-схемы, то остается определить, как строить управляющие процессоры прямого и обратного просмотров. Далее следует запись алгоритма построения управляющего процессора на алголоподобном[69] языке с комментариями, которые в некоторых случаях заменяют операторы. Последние в отличие от настоящих комментариев, заключаемых в специальные ограничители (#), ничем не выделяются.

Построение управляющего процессора прямого просмотра.

Вход: $\mathscr{G}_c$ = (N, T, $\Re$, Δ, K, *S*) — управляющая граф-схема

Выход: $\mathscr{P}_c^f$ =(Q^f, Σ^f, Γ^f, Δ^f, $\Re^f$, δ^f, q_0^f, F^f) — управляющий процессор прямого просмотра

Метод:

Инициализация

$\Sigma^f := T$; # T — алфавит терминалов управляющей граф-схемы $\mathscr{G}_c$ #

Δ^f — такой же как в управляющей граф-схеме[70];

$q_0^f := \{v_0 \mid v_0 \in \mathrm{Vertex}(\mathscr{G}_c) : (\mathrm{Mark}(v_0) = \text{"begin-S"})\}$;

$Q^f := \{ q_0^f \}$; $\Gamma^f := \varnothing$; $\Re^f := \varnothing$; $F^f := \varnothing$; # Эти четыре множества в дальнейшем пополняются #

Continue := **true**;

Главный цикл построения управляющего процессора прямого просмотра состоит из цикла совместного пополнения таблиц δ_1^f и δ_2^f, и затем цикла пополнения таблицы δ_3^f, исполняемых поочередно до тех пор, пока флажок Continue не перейдет в состояние **false**. Это происходит тогда, когда в результате очередного выполнения этапа пополнения таблицы δ_3^f не появляется ни одного нового возвратного состояния. И тогда производится *проверка внешней балансировки* управляющей граф-схемы. Содержательный смысл этой проверки — обнаружение ситуаций, в которых не ясно, отнести ли текущий входной символ к текущей или к объемлющей конструкции языка. Если такая проверка не обнаруживает ни одной неясной ситуации, то гарантирована детерминированность процессора прямого просмотра относительно альтернативы: использовать верхний символ магазина или нет.

---

[69] Имеется в виду, что используется по возможности синтаксис языка Алгол 68.

[70] Напомним, что $\Delta = \Delta^f \cup \Delta^b$, $\Delta^f \cap \Delta^b = \varnothing$.

while Continue
do Continue := **false**; Continue2 := **false**;

Пополнение таблиц δ_1^f и δ_2^f.

На этом этапе производится разложение состояний, имеющихся в множестве Q^f. По разложению состояния проверяются необходимые условия детерминированности процессора и планируются все его возможные движения в данном состоянии. Появляющиеся при этом новые состояния включаются в множество Q^f [71], а новые магазинные символы — в множество Γ^f.

Этап пополнения таблиц δ_1^f и δ_2^f заканчивается, когда рассмотрены все состояния из Q^f. #

for $\forall q : (q \in Q^f)$
do D := Develope(q); # Разложение состояния q — лес, состоящий из деревьев, корнями которых являются вершины множества q. #
if Height(D) = ∞ [72]
then Alarm(("Разложение состояния ", q, " бесконечно!"))
else # Построение непустых движений в состоянии q. #
for $\forall a : (a \in \Sigma^f \& \exists v : (v \in \text{Leaves}(D) \& \text{Mark}(v) = a))$
do # Отбор ветвей разложения D, листья которых помечены входным символом a. #
D_a := Subforest(D, a);
if $D_a \neq \varnothing$
then $R := \varnothing$; # Сброс резольверных входов для символа a. #
for $\forall r : (r \in \Re^{f^*} \& \exists b : (b \in \text{Branches}(D_a) \& \text{Mark}^{\Re^f}(b) = r))$
do # Отбор ветвей из D_a, метки которых, если в них учитывать только резольверные символы прямого просмотра, образуют r. #
$D_{a,r}$ = Selected branches(D_a, r);
if Не все ветви $D_{a,r}$ имеют равную длину
then Alarm(("Нарушена балансировка разложения " "состояния ", q, " по входному " "символу ", a, " и составному " "резольверу ", r, "!"))
else # Обеспечена детерминированность уровня конструкции, к которой относится текущий входной символ #
σ := Forward pass semantics ($D_{a,r}$);

---

[71] Те из них, которые оказываются конечными, пополняют также множество F^f.

[72] Разумеется, эта запись условна. Бесконечные разложения появляются тогда и только тогда, когда исходная управляющая грамматика леворекурсивна.

if Ambiguous(σ)
then Alarm(("Семантическая неоднозначность "
"состояния ", q, " по входному "
"символу ", a, " и составному "
"резольверу ", r, "!"))
else $R := R \cup \{r\}$; # Пополнение резольверных входов для a #
$p :=$ Leaves($D_{a,r}$); # p — переходное состояние из q по входному символу a при условии, что выполняется составной предикат r, т.е. если $\iota(r)$. #
$Q^f := Q^f \cup \{p\}$;
α := Push down list($D_{a,r}$); # α — магазинная цепочка. Если она содержит символы, которых еще нет в алфавите Γ^f, то Push down list устанавливает флажок Continue2 в состояние **true.** #
$\delta_2^f(q, a, r) := (p, \alpha, \sigma)$
fi
fi
od; # Конец цикла определения резольверных входов для a #
$\delta_1^f(q, a) := R$
fi # Конец определения движения для входного символа a #
od; #Конец цикла построения непустых движений в состоянии q #
Построение ε–движений в состоянии q.
$D_f := \bigcup_{A \in N}$ Subforest(D, "end-A");
D_f — те ветви разложения D, листья которых помечены символами вида "end-A", где A — некоторый нетерминал.
if $D_f \neq \varnothing$
then # Пустые движения существуют
(q — подавляемое состояние). #
$R := \varnothing$; # Сброс резольверных входов для символа ε. #
for $\forall r : (r \in \Re^{f^*} \,\&\, \exists b : (b \in \text{Branches}(D_f) \,\&\, \text{Mark}^{\Re^f}(b) = r))$
do # Отбор ветвей D_f, метки которых, если в них учитывать только резольверные символы прямого просмотра, образуют цепочку r. #
$D_{f,r}$:= Selected branches($D_{f,r}$);
if Не все ветви $D_{f,r}$ имеют равную длину
then Alarm(("Нарушена балансировка "
"разложения состояния ", q,
" по конечным меткам и "
"составному резольверу ", r, "!"))
else $\sigma :=$ Forward pass semantics ($D_{f,r}$);

```
            if Ambiguous(σ)
            then  Alarm(("Семантическая неоднозначность "
                  "состояния ", q, "по конечным вершинам и "
                   "составному резольверу ", r, "!" ))
            else
                 if Mask(q, r) = {S}[73]
                 then # q — конечное состояние #
                      F^f := F^f ∪ {q}
                 else # q — подавляемое состояние #
                      Continue2 := true
                 fi;
                    R := R ∪ {r}; # Пополнение резольверных входов для ε #
                    α := Push down list(D_{f,r});
                    # α — магазинная цепочка. Если она содержит магазинные
                    символы, которых еще нет в алфавите Γ^f, то Push down list
                    устанавливает флажок Continue2 в состояние true. #
                                    δ2^f( q, ε, r) := (Sup, α, σ);
              fi
            fi
          od; # Конец цикла определения резольверных входов для ε.#
                    δ1^f(q, ε) := R
                    fi # Конец определения ε -движений в подавляемом
                  состоянии q. #
        fi
      fi

      od ; # Конец этапа пополнения таблиц δ1^f и δ2^f.#
```

Пополнение таблицы δ_3^f.

Этот этап выполняется только при условии, что флажок Continue2 был установлен в состояние **true** в цикле пополнения таблиц δ_1^f и δ_2^f. Это же случается тогда, когда во время выполнения упомянутого цикла образуется новое подавляемое состояние или новый магазинный символ.

Рассматриваются всевозможные тройки вида (q, X, r), где q — подавляемое состояние, $X \in \Gamma^f$, $r \in (\Re^{f*} \cup \{\varepsilon\})$, для которых значение $\delta_3^f(q, X, r)$ не определено, тогда как значение $\delta_2^f(q, \varepsilon, r)$ определено. Если q и X сопряжены по составному резольверу r (предполагается, что $\iota(r) = \textbf{true}$, когда $r = \varepsilon$), то по ним определяется возвратное состояние s, которое пополняет множество Q^f, и $\delta_3^f(q, X, r) := s$.

---

[73] Предполагается, что начальный нетерминал S не встречается в правых частях правил грамматики. Во всяком случае, путем простого эквивалентного преобразования грамматики этого легко добиться.

Цикл пополнения таблицы возвратных состояний δ_3^f заканчивается, когда рассмотрены все упомянутые тройки.

Если во время выполнения этого цикла появляются новые состояния, то после его завершения снова повторяется этап пополнения таблиц δ_1^f и δ_2^f. #

while Continue2
do Continue2 := **false;**
for $\forall q$: ($q \in Q^f$ & Suppressible(q))
do
for $\forall X$: ($X \in \Gamma^f$)
do
for $\forall r$: ($r \in (\Re^{f*} \cup \{\varepsilon\})$)
do
if $\delta_3^f(q, X, r)$ = **undefined** & Conj(q, X, r)
then # q и X сопряжены по составному резольверу r. #
$s := \{v | v \in X \& \text{Mark}(v) \in \text{Mask}(q, r)\}$; $\delta_3^f(q, X, r) := s$;
if $s \notin Q^f$
then # s — новое состояние. #
$Q^f := Q^f \cup \{s\}$; Continue := **true**
fi
fi
od # r #
od # X #
od # q #
od # Конец цикла пополнения таблицы возвратных состояний. #
od; # Конец главного цикла. #
На этом заканчивается построение управляющей таблицы прямого просмотра. Затем производится последняя проверка, а именно: проверка внешней балансировки грамматики (граф-схемы).

Проверка внешней балансировки. #

Ok := **true**;
for $\forall q$: ($q \in Q^f$ & Suppressible(q)) **while** Ok
do # Проверяется условие внешней балансировки для каждого дерева возвратных состояний, построенного для подавляемого состояния q. #
Ok := External balance(Return tree(q));
if $\neg$Ok
then Alarm ("Нарушена внешняя балансировка "
"в состоянии ", q)
fi
od

Если во время исполнения последнего цикла не обнаружено нарушение внешней балансировки[74], то построенная управляющая таблица прямого просмотра δ^f является правильной. Иначе она не годится для использования и в этом случае необходимо попытаться исключить обнаруженные дефекты путем эквивалентных преобразований исходной грамматики.

Вспомогательные алгоритмы для построения прямого просмотра. Далее кратко опишем вспомогательные процедуры, использующиеся в рассмотренном алгоритме построения прямого просмотра.

Alarm — сигнализация нарушения ограничений, накладываемых на грамматику. Процедура выдает сообщение о нарушении необходимых и достаточных условий, гарантирующих существование процессора требуемого класса. Она же прекращает процесс его построения.

Ambiguous — семантическая неоднозначность. Процедура дает результат **true**, если множество, заданное ее параметром, содержит более одной семантической цепочки. В противном случае ее результат — **false**.

Branches — множество ветвей, составляющих данный лес. Процедура Branches(f), где f — некоторый лес, выдает множество ветвей, его составляющих.

Conj — сопряженность подавляемого состояния с магазинным символом. Считается, что подавляемое состояние $q \in Q^f$ *сопряжено* с магазинным символом $X \in \Gamma^f$ при условии r, если в момент обращения процессора прямого просмотра к магазину в состоянии q, когда интерпретация резольверной цепочки r дает **true**, на его вершине может оказаться символ X.

Функция Conj(q, X, r), где q — подавляемое состояние, X — магазинный символ, r — составной резольвер, определяется следующим образом:

$\mathrm{Conj}(q, X, r) \stackrel{def}{=} \mathrm{Mask}(q, r) \subseteq \mathrm{Mark}(X)$[75].

Create edge — образовать ребро. Процедура Create edge(u, v) образует ориентированную дугу от вершины u к вершине v.

Develope — разложение состояния. Разложение состояния $q \in Q^f$, являющегося параметром процедуры Develope, выполняется по следующим шагам:

1. Создаются копии вершин, входящих в множество q. Будем считать, что они принадлежат нулевому уровню разложения данного состояния q. Далее эти копии вершин считаются корнями деревьев, построение которых производится при помощи последующих шагов алгоритма.

2. К каждому корню пристраиваются копии дуг управляющей граф-схемы, инцидентных его прообразу, вместе с их концевыми вершинами и контекстными[76] метками, помечающими эти дуги. Будем считать, что множество присоединенных таким образом вершин образует первый уровень разложения состояния q. Далее, level := 1.

---

[74] Балансировка называется *внешней*, так как она не может быть проверена локально по разложению состояния, а выполняется с учетом глобальной информации, содержащейся в управляющей таблице прямого просмотра.

[75] См. определение функций Mask и Mark далее в этом разделе.

[76] Т.е. семантическими и резольверными.

3. К каждой вершине уровня level, которая помечена нетерминалом, пристроим копии дуг управляющей граф-схемы, инцидентных начальной вершине компоненты, определяющей этот нетерминал, вместе с концевыми вершинами и контекстными метками, помечающими эти дуги. Вновь присоединенные вершины будем считать относящимися к уровню level + 1 разложения состояния q. Далее, level := level + 1.

Шаг 3 повторяется до тех пор, пока не обнаружится, что на текущем уровне нет ни одной нетерминальной вершины, и тогда алгоритм построения разложения состояния q заканчивается. Если же шаг 3 "зацикливается", разложение состояния q — бесконечно[77].

External balance — проверка внешней балансировки. Процедура External balance проверяет выполнение условия внешней балансировки для дерева возвратных состояний, корень которого задан как ее параметр. Это условие состоит в том, что для каждой пары вершин, принадлежащих одной и той же ветви этого дерева и представляющих некоторые состояния q_1 и q_2, должно быть:

$$\mathrm{Accept}(q_1) \cap \mathrm{Accept}(q_2) = \varnothing,$$

где $\mathrm{Accept}(q) \stackrel{def}{=} \{a \mid a \in \Sigma^{\mathrm{f}} \ \& \ \exists r\colon \ (r \in \Re^{\mathrm{f}^*} \& \ \delta_2^{\mathrm{f}}(q,\ a,\ r)$ — определено$\}$ — множество входных символов, допустимых в состоянии q.

Forward pass semantics — семантика леса. Процедура Forward pass semantics(f) определяет множество семантических цепочек прямого просмотра для леса f, заданного ее параметром. Именно, для каждой ветви леса f определяется цепочка, составленная из семантических символов прямого просмотра, помечающих эту ветвь. Множество всех таких цепочек выдается в качестве результата процедуры.

Height — высота леса. Процедура Height вычисляет высоту леса, заданного ее параметром. Она равна длине самой длинной ветви по всем деревьям, входящим в данный лес, и выражается числом дуг, составляющих эту ветвь.

Leaves — множество листьев. Процедура Leaves выдает множество листьев, т. е. концевых вершин леса или множества ветвей, заданного ее параметром.

Mark — метки вершин. Процедура Mark выдает множество меток, помечающих вершины, заданные ее параметром.

Mark$^{\Re^{\mathrm{f}}}$ — резольверные метки. Процедура выдает цепочку резольверов прямого просмотра, помечающих данную ветвь разложения состояния прямого просмотра.

Mask — нетерминалы меток конечных вершин.

Процедура Mask$(q,\ r)$, где q — подавляемое состояние, r — цепочка резольверных символов прямого просмотра, выдает множество нетерминалов, входящих в состав меток конечных вершин — листьев тех ветвей разложения

[77] Разумеется, фактическое "зацикливание" алгоритма не допускается. Признаком, предупреждающим о зацикливании, является появление на какой-нибудь ветви разложения двух вершин, помеченных одинаковыми нетерминальными символами.

состояния q, которые помечены цепочкой r, если в метках этих ветвей игнорировать все семантические символы и резольверные символы обратного просмотра, т. е.

$$\begin{aligned}\mathrm{Mask}(q,\ r) \stackrel{def}{=} \{A \mid A \in \mathrm{N}\ \&\\ \&\ \exists v : (v \in \mathrm{Leaves}(\mathrm{Selected\,branches}(\mathrm{Develope}(q), r))\ \&\\ \&\ \mathrm{Mark}(v) = \text{"end-}A\text{"})\}.\end{aligned}$$

Push-down list — магазинная цепочка. Эта процедура выдает цепочку магазинных символов, которая строится по лесу, заданному ее параметром. Каждый новый магазинный символ, представляемый множеством нетерминальных вершин одного уровня, вносится в словарь магазинных символов Γ^f. Вот алгоритм ее реализации.

Вход: f — лес, некоторое множество ветвей разложения некоторого состояния. Все ветви f имеют одинаковую длину.

Выход: α — цепочка магазинных символов из Γ.

Метод: $\alpha := \varepsilon$;

for i **to** $\mathrm{Height}(f) - 1$

do # Инициализация магазинной цепочки #

$X := \{v \mid v \in Vertex_i\ (f)\}$; # X — магазинный символ, множество нетерминальных вершин f уровня i. #

$\alpha := X\alpha$; # Наращивание магазинной цепочки #

if $X \notin \Gamma$

then # Пополнение алфавита магазинных символов #

$\Gamma := \Gamma \cup \{X\}$

fi

od

Return tree — дерево возвратных состояний. Эта процедура строит дерево возвратных состояний для подавляемого состояния q, заданного в качестве ее параметра. Именно, строится корень дерева $R = \mathrm{Repr}(q)$, представляющий данное состояние q, и инициализируется дерево возвратных состояний: $\mathrm{RT} := R$. Считается, что R принадлежит уровню 0.

Далее к каждому листу дерева, представляющему подавляемое состояние, пристраиваются вершины, представляющие все возможные возвратные состояния для данного подавляемого состояния (которые зависят также от резольверных цепочек и магазинных символов).

Процесс заканчивается, когда в конструируемом дереве все листья представляют неподавляемые состояния или являются повторными образами подавляемых состояний, уже имеющихся на предыдущих уровнях дерева[78]. Для этого выполняются следующие действия:

[78] В приведенном далее алгоритме появление повторных образов подавляемых состояний не контролируется. Поэтому данный алгоритм "зацикливается", если дерево возвратных состояний оказывается бесконечным.

for i **from** 0 **while** $\exists v:(v\in \mathrm{Vertex}_i(\mathrm{RT})\,\&$

$\exists p:(v=\mathrm{Repr}(p)\,\&\,\mathrm{Suppressible}(p)))$

do for $\forall v:(v\in \mathrm{Vertex}_i(\mathrm{RT})\,\&\,\exists p:(v=\mathrm{Repr}(p)\,\&\,\mathrm{Suppressible}(p)))$

Здесь $\mathrm{Vertex}_i(\mathrm{RT})$ — множество вершин дерева возвратных состояний RT уровня i. Равенство $v=\mathrm{Repr}(p)$ означает, что вершина v дерева возвратных состояний RT представляет состояние p.

do for $\forall X:(X\in \Gamma_{\mathrm{f}})$

do for $\forall r:(r\in \mathfrak{R}^{\mathrm{f}^*}\,\&\,\delta_3^{\mathrm{f}}(p,X,r)=q)$

do # Провести дугу из вершины v во вновь создаваемую вершину w, которая представляет возвратное состояние q. Вершину w считать относящейся к уровню $i+1$ #

w := *Repr*(q); Create edge(v, w)

od

od

od

od

Selected branches — множество ветвей, помеченных данной резольверной цепочкой. Процедура Selected branches(f, r), где f — некоторый лес, а r — цепочка резольверных символов прямого просмотра, выдает множество ветвей f, метки которых, если в них игнорировать все семантические символы и резольверные символы обратного просмотра, равны r.

Suppressible — тестирование подавляемого состояния. Процедура Suppressible(q), где q — некоторое состояние, выдает **true**, если состояние q — подавляемое, и **false** — в противном случае.

Vertex$_i$ — множество вершин уровня i. Процедура $\mathrm{Vertex}_i(f)$ выдает множество вершин уровня i в дереве или лесу f, заданном ее параметром.

Построение управляющего процессора обратного просмотра:

Вход: $\mathscr{G}_{\mathrm{c}}$ = (N, T, $\mathfrak{R}$, Δ, K, S) — управляющая граф-схема;
Q^{f} — множество состояний прямого просмотра[79]

Выход: $\mathscr{R}_{\mathrm{c}}^{\mathrm{b}} = (\mathrm{Q}^{\mathrm{b}},\Sigma^{\mathrm{b}},\Gamma^{\mathrm{b}},\Delta^{\mathrm{b}},\mathfrak{R}^{\mathrm{b}},\delta^{\mathrm{b}}, q_0^{\mathrm{b}},\mathrm{F}^{\mathrm{b}})$ — искомый управляющий процессор обратного просмотра, где Q^{b} — множество состояний, Σ^{b} — входной алфавит, Γ^{b} — магазинный алфавит, Δ^{b} — алфавит семантик, $\mathfrak{R}^{\mathrm{b}}$ — алфавит резольров, $\delta^{\mathrm{b}}=(\delta_1^{\mathrm{b}},\delta_2^{\mathrm{b}})$ — управляющая таблца, состоящая из таблицы резольверов $\delta_1^{\mathrm{b}}:\mathrm{Q}^{\mathrm{b}}\times(\Sigma^{\mathrm{b}}\cup\{\varepsilon\})\to 2^{\mathfrak{R}^{\mathrm{b}^*}}$ и таблицы управляющих элементов $\delta_2^{\mathrm{b}}:\mathrm{Q}^{\mathrm{b}}\times(\Sigma^{\mathrm{b}}\cup\{\varepsilon\})\times\mathfrak{R}^{\mathrm{b}^*}\to(\mathrm{Q}^{\mathrm{b}}\cup\{\mathbf{Pop}\})\times(\Gamma^{\mathrm{b}}\cup\{\varepsilon\})\times\Delta^{\mathrm{b}^*}$, q_0^{b} — начальное состояние, $\mathrm{F}^{\mathrm{b}}\subseteq\mathrm{Q}^{\mathrm{b}}$ — множество конечных состояний.

---

[79] Напомним, что каждое состояние управления представляется как некоторое множество вершин управляющей граф-схемы.

Метод:

Инициализация

$\Sigma^b := Q^f$; Δ^b и $\Re^b$ — как в управляющей граф-схеме;

$q_0^b := \{v \mid v \in \mathrm{Vertex}(\mathcal{G}_c) : \mathrm{Mark}(v) = \text{"end-}S\text{"}\}$;

$F^b := \varnothing$; $Q^b := \{ q_0^b \}$; $\Gamma^b := \varnothing$;

Построение δ^b

for $\forall q : (q \in Q^b)$ # q — некоторое состояние обратного просмотра #

do

for $\forall t : (t \in \Sigma^b)$ # t — некоторое состояние прямого просмотра #

do $E := \mathrm{Edges}(t, q)$; # E — множество дуг граф-схемы, начала которых находятся в множестве t, а концы — в множестве q #

if $E \neq \varnothing$

then # E — не пусто #

$R :=$ Backward pass resolvers(E); # R — множество цепочек, состоящих из резольверных символов обратного просмотра, входящих в состав контекстных цепочек, помечающих дуги множества E #

$\delta_1^b(q, t) := R$;

for $\forall r : (r \in R)$

do

SE := Select edges(E, r); # SE — подмножество дуг E, метки которых, если в них принимать во внимание только резольверные символы обратного просмотра, равны r #

$p := \varnothing$;

for $\forall e \in \mathrm{SE}$ **do** $p := p \cup \mathrm{Start}(e)$ **od** ;

Определение семантики

σ := Backward pass semantics(SE);

if Ambiguous(σ)

then Alarm ("Семантическая неоднозначность состояния ", q, "по входу ", t, " и резольверу ", r)

fi;

if Nonterminals(p)

then # p — нетерминальные вершины: прохождение правой границы терминального порождения некоторого понятия, помечающего одну из вершин множества p. #

$s := \{v \mid v \in \mathrm{Vertex}(\mathcal{G}_c) \,\&\, \mathrm{Mark}(v) = \text{"end-}A\text{"} \,\&\, A \in \mathrm{Mark}(p)\}$;

if $s \notin Q^b$

then # s — новое состояние #

$Q^b := Q^b \cup \{s\}$

fi;

if $p \notin \Gamma^b$
then # p — новый магазинный символ #
$\Gamma^b := \Gamma^b \cup \{p\}$; $Q^b := Q^b \cup \{p\}$;
fi;
$\delta_2^b(q, t, r) := (s, p, \sigma)$
else # p состоит из терминальных вершин или начальной вершины заглавной компоненты управляющей граф-схемы #
$\delta_2^b(q, t, r) := (p, \varepsilon, \sigma)$
fi
od # r #
fi
od # t #;
$s := \{v \mid v \in \mathrm{Vertex}(\mathcal{G}_c) \,\&\, \mathrm{Mark}(v) = \text{"begin-}A\text{"} \,\&\, \mathrm{Succ}(v) \cap q \neq \varnothing\}$;
if $s \neq \varnothing$
then # Достигнута левая граница терминального порождения некоторого понятия, являющегося меткой одной из вершин, входящих в множество, представляемое верхним символом магазина. #
E := Edges (s, q); # E — множество дуг граф-схемы, начала которых находятся в множестве s, а концы — в множестве q #
R := Backward pass resolvers (E); # R — множество цепочек, состоящих из резольверных символов обратного просмотра, входящих в состав контекстных цепочек, помечающих дуги множества E #
$\delta_1^b(q, \varepsilon) := R$;
for $\forall r : (r \in R)$
do SE := Select edges(E, r); # SE — подмножество дуг E, метки которых, если в них принимать во внимание только резольверные символы обратного просмотра, равны r #
Определение семантики
σ := Backward pass semantics(SE);
if Ambiguous (σ)
then # Диагностика семантической неоднозначности #
Alarm ("Семантическая неоднозначность состояния ", q,
" по ε–входу и резольверу ", r)
fi;
$\delta_2^b(q, \varepsilon, r) := (\mathbf{Pop}, \varepsilon, \sigma)$
od # r #
od # q #;
$F^b := \{q \in Q^b \mid \mathrm{Edges}(\{v_0\}, q) \neq \varnothing, \mathrm{Mark}(v_0) = \text{"begin-}S\text{"}\}$
Конечными являются те состояния обратного просмотра, которые содержат вершины, являющиеся концами дуг, инцидентных начальной вершине заглавной компоненты управляющей граф-схемы.

На этом заканчивается построение управляющего процессора обратного просмотра, а вместе с тем и построение всего челночного процессора.

Вспомогательные алгоритмы для построения обратного просмотра. Здесь мы коротко опишем вспомогательные процедуры, использующиеся только при построении обратного просмотра.

Backward pass resolvers — резольверы обратного просмотра. Процедура Backward pass resolvers(E), где E — некоторое множество дуг управляющей граф-схемы, дает множество цепочек резольверов обратного просмотра, заложенных[80] в метки этих дуг.

Backward pass semantics — семантики обратного просмотра. Процедура Backward pass semantics(E), где E — некоторое множество дуг управляющей граф-схемы, выдает множество семантических цепочек обратного просмотра, заложенных[81] в метки дуг множества E.

Edges — дуги. Процедура Edges(p, q), где p и q — некоторые множества вершин управляющей граф-схемы, выдает множество дуг, начала которых находятся в множестве p, а концы — в множестве q.

Nonterminals — нетерминалы. Процедура дает результат **true**, если множество, заданное ее параметром, состоит из нетерминальных вершин, и **false** — в противном случае.

Select edges — выборка подмножества дуг. Процедура Select edges(E, r) из множества дуг E отбирает те, которые помечены контекстными цепочками, в которые заложена цепочка резольверных символов обратного просмотра r.

Start — начало дуги. Процедура Start (e), где e — дуга, дает вершину, из которой эта дуга выходит.

Все другие процедуры, используемые в алгоритме построения обратного просмотра, но здесь не определенные, были описаны ранее.

3.5. СПЕЦИФИКАЦИЯ ЧЕЛНОЧНЫХ ТРАНСЛЯЦИЙ ПРИ ПОМОЩИ ТРАНСЛЯЦИОННЫХ RBNF-ГРАММАТИК

В заключение этой главы определим, каким образом челночные трансляции можно специфицировать и при помощи трансляционных RBNF-грамматик.

Очевидно, что словари контекстных символов управляющей грамматики должны быть разделены по просмотрам. Термин *просмотр* в данном случае будет относится к порядку интерпретации управляющих цепочек, порождаемых управляющей грамматикой. Модифицированные таким образом трансляционные грамматики также будем называть *челночными*.

---

[80] Имеются в виду те цепочки, которые получаются из меток дуг, о которых идет речь, если в них игнорировать все символы, кроме резольверов обратного просмотра.

[81] Имеются в виду те цепочки, которые получаются из меток дуг, о которых идет речь, если в них игнорировать все символы, кроме семантик обратного просмотра.

Нет нужды определять челночные трансляционные грамматики заново. Достаточно дополнить определение трансляционной грамматики, приведенное в разд. 1.2, указанием, что $\Re = \Re^{f} \cup \Re^{b}$, $\Re^{f} \cap \Re^{b} = \varnothing$, $\Sigma = \Sigma^{f} \cup \Sigma^{b}$, $\Sigma^{f} \cap \Sigma^{b} = \varnothing$, и принять во внимание расширенное понимание операционной среды, данное в разд. 3.2. Тогда под *трансляцией, специфицируемой челночной трансляционной грамматикой* G_S, будем понимать множество

$\tau(G_S) = \{(x, [e]_H) | \exists C_x : (C_x \in \mathscr{C}(G_C), C_x = \kappa_0 a_1 \kappa_1 a_2 ... \kappa_{n-1} a_n \kappa_n$, где $a_i \in T$ $(i = 1, 2, ..., n;$ $n \geq 0)$, $\kappa_m \in (\Re \cup \Sigma)^*$ $(m = 0, 1, 2, ..., n)$ интерпретируются синхронизированно, $x = a_1 a_2 ... a_n$ — входная цепочка$\}$.

Синхронизированная интерпретация контекстных символов, распределенных по просмотрам, определяется аналогично описанной в разд. 1.2. Пусть ρ_m^{f} и ρ_m^{b} — резольверные подцепочки прямого и обратного просмотров контекстной цепочки κ_m, σ_m^{f} и σ_m^{f} — семантические подцепочки прямого и обратного просмотров контекстной цепочки κ_m[82]. Синхронизированная интерпретация контекстных символов предполагает, что выполняются следующие условия и имеют место соответствующие равенства:

На прямом просмотре:	На обратном просмотре:
$\iota_{\rho_0^{f}}(e_0) : e_1^{f} = \iota_{\sigma_0^{f}}(e_0)$,	$\iota_{\rho_n^{b}}(e_0^{b}) : e_1^{b} = \iota_{\sigma_n^{b}}(e_0^{b})$,
$\iota_{\rho_1^{f}}(e_1^{f}) : e_2^{f} = \iota_{\sigma_1^{f}}(e_1^{f})$,	$\iota_{\rho_{n-1}^{b}}(e_0^{b}) : e_2^{b} = \iota_{\sigma_{n-1}^{b}}(e_1^{b})$,
...	...
$\iota_{\rho_n^{f}}(e_n^{f}) : e_0^{b} = \iota_{\sigma_n^{f}}(e_n^{f})$.	$\iota_{\rho_0^{b}}(e_n^{b}) : e = \iota_{\sigma_0^{b}}(e_n^{b})$.

Именно, сначала вычисляется конъюнкция предикатов, ассоциированных с резольверными символами прямого просмотра, предшествующими первому терминалу в управляющей цепочке (ρ_0^{f}). Эти предикаты вычисляются на начальном состоянии операционной среды (e_0). Если все они выполняются, то производятся преобразования начального состояния операционной среды (e_0), ассоциированные с семантическими символами прямого просмотра, предшествующими первому терминалу, в текстуальной последовательности (σ_0^{f}). Результатом этих преобразований является новое состояние операционной среды (e_1^{f}).

Затем на достигнутом состоянии операционной среды (e_1^{f}) вычисляются предикаты, ассоциированные с резольверными символами прямого просмотра, предшествующими второму терминалу в управляющей цепочке (ρ_1^{f}). Если все они выполняются, производятся преобразования текущего состояния операционной среды (e_1^{f}), ассоциированные с семантическими символами прямого просмотра, предшествующими второму терминалу, в текстуальной последовательности (σ_1^{f}). Результатом этих преобразований является новое состояние операционной среды (e_2^{f}), и т. д.

[82] Мы используем обозначения, введенные при определении трансляции $\tau(G_s)$.

Когда все контекстные символы прямого просмотра таким синхронизированным образом будут проинтерпретированы, начинается интерпретация контекстных символов обратного просмотра в следующем порядке: сначала вычисляется конъюнкция предикатов, ассоциированных с резольверными символами обратного просмотра, следующими за последним терминалом управляющей цепочки (ρ_n^b). Если она выполняется, производятся преобразования операционной среды, ассоциированные с семантическими символами обратного просмотра, следующими за последним терминалом (σ_n^b), в порядке, обратном их вхождению в управляющую цепочку, и т. д. Если, наконец, вычисление конъюнкции предикатов, ассоциированных с резольверными символами обратного просмотра, предшествующими первому терминалу (ρ_0^b), дает истину, то выполняются преобразования операционной среды, ассоциированные с семантическими символами обратного просмотра, предшествую-щими первому терминалу (σ_0^b), в обратном порядке.

Полученное финальное состояние операционной среды (e) в проекции на объектное подпространство H, и есть результат челночной трансляции входной цепочки. Если же хотя бы один предикат, вычисляемый в этом процессе, не выполняется, то такая управляющая цепочка отвергается, но это еще не значит, что входная цепочка содержит контекстную ошибку. Входная цепочка будет содержать контекстную ошибку, если не существует никакой управляющей цепочки, терминальные символы которой образуют входную цепочку, все предикаты которой синхронизированно выполняются.

Глава 4

ОПТИМИЗАЦИЯ ЧЕЛОЧНЫХ ПРОЦЕССОРОВ

4.1. О МЕТОДЕ ОПТИМИЗАЦИИ ЧЕЛНОЧНЫХ ПРОЦЕССОРОВ

Управляющие таблицы челночных процессоров, способ построения которых был описан в предыдущей главе, в некоторых случаях могут получиться весьма объемными. Путем искусного кодирования можно добиться некоторой экономии памяти, необходимой для их размещения. И хотя эту возможность не следует упускать из виду в любом случае, существует принципиально другой способ для оптимизации челночных процессоров, который состоит в определении отношений эквивалентности на множествах состояний, входных и магазинных символов и использовании в управляющих таблицах классов эквивалентности в этих отношениях взамен элементов соответствующих множеств. Т.е. вместо состояния, входного или магазинного символа на входе управляющей таблицы или в управляющем элементе следует использовать соответствующий класс эквивалентности, которому это состояние, этот входной или мАгазинный символ принадлежит. Благодаря такой подмене число входов в управляющие таблицы значительно сокращается, а их объем, как показывает практика, уменьшается на десятки процентов, а то и в несколько раз. При этом заметно сокращается количество обращений к таблице возвратных состояний, и часто она оказывается совсем ненужной.

Опишем подробнее этот метод оптимизации челночных процессоров, который применим и к однопросмотровым процессорам, как сплайновым, так и конечным.

Отношения эквивалентности, о которых идет речь, определяются на основании принципа неразличимости соответственно состояний, входных и магазинных символов по реакции управляющих процессоров.

4.2. ЭКВИВАЛЕНТНОСТЬ СОСТОЯНИЙ, МАГАЗИННЫХ И ВХОДНЫХ СИМВОЛОВ

***Определение* 4.2.1.** Пусть q_1, $q_2 \in Q^f$ — два состояния прямого просмотра. Состояния q_1 и q_2 считаются *различными*, если выполняется хотя бы одно из следующих условий:

1) различны множества входных символов (включая ε[83]), допустимых в этих состояниях, т. е. существует входной символ $a \in \Sigma^f \cup \{\varepsilon\}$, такой, что из двух значений $\delta_1^f(q_1, a)$ и $\delta_1^f(q_2, a)$ определено только одно; иначе

2) существует такой входной символ $a \in \Sigma^f \cup \{\varepsilon\}$, для которого множества контекстов, в которых он допустим в рассматриваемых состояниях, различны, т. е. $\delta_1^f(q_1, a) \neq \delta_1^f(q_2, a)$; в противном случае

3) существуют такой входной символ $a \in \Sigma^f \cup \{\varepsilon\}$и такая резольверная цепочка $r \in \Re^{f*}$, для которых соответствующие управляющие элементы различаются семантическими или магазинными цепочками, т. е. если $\delta_2^f(q_1, a, r) = (p_1, \alpha_1, \sigma_1)$ и $\delta_2^f(q_2, a, r) = (p_2, \alpha_2, \sigma_2)$, то $\sigma_1 \neq \sigma_2$ и/или $\alpha_1 \neq \alpha_2$ (см. далее определение 4.2.3); в противном случае

4) если оба состояния q_1 и q_2 — подавляемые, различны множества магазинных символов, сопряженных с данными состояниями по резольверной цепочке r, т. е. существуют магазинные символы $X, Y \in \Gamma^f$, такие, что из двух значений $\delta_3^f(q_1, X, r)$ и $\delta_3^f(q_1, Y, r)$ определено только одно; в противном случае эти два возвратных состояния различаются по данному определению;

5) рассматриваемые состояния q_1 *и* q_2 различны как входные символы обратного просмотра, если он используется; либо

6) различны соответствующие переходные состояния p_1 и p_2 в силу данного определения[84].

***Определение* 4.2.2.** Состояния прямого просмотра q_1, $q_2 \in Q^f$ *эквивалентны*, если они не различимы согласно определению 4.2.1.

Эквивалентность состояний обратного просмотра определяется аналогично с той лишь разницей, что это делается безотносительно к управляющей таблице прямого просмотра и без ссылки на таблицу возвратных состояний, которая на обратном просмотре не используется.

***Определение* 4.2.3.** Магазинные цепочки $\alpha_1 = X_1 X_2 \ldots X_m$ и $\alpha_2 = Y_1 Y_2 \ldots Y_n$ ($X_i, Y_j \in \Gamma^f$, $1 \le i \le m$, $m \ge 0$, $1 \le j \le n$, $n \ge 0$) — *различны*, если различна их длина, т. е. $m \neq n$, в противном случае, если существует такое i ($1 \le i \le m$), что магазинные символы X_i *и* Y_i — различны в силу следующего определения:

***Определение* 4.2.4.** Два магазинных символа $X, Y \in \Gamma^f$ — *различны*, если выполняется хотя бы одно из следующих условий:

1) существует резольверная цепочка $r \in \Re^{f*}$, такая, что различны множества подавляемых состояний, сопряженных с этими символами по составному резольверу r, в противном случае

---

[83] Строго говоря, ε — не входной символ, а цепочка, в которой нет ни одного символа.

[84] В этом определении первые пять признаков различимости — базовые, а шестой определяет рекурсивное сведение к этим базовым признакам.

2) если существует такое подавляемое состояние $q \in Q^f$, сопряженное с каждым из этих магазинных символов по некоторому составному резольверу $r \in \Re^{f*}$, для которого таблица возвратных состояний дает различающиеся (согласно определению 4.2.1) возвратные состояния $p_1 = \delta_3^f(q, X, r)$ *и* $p_2 = \delta_3^f(q, Y, r)$.

Определение **4.2.5.** Два магазинных символа $X \in \Gamma^f$ *и* $Y \in \Gamma^f$ — *эквивалентны*, если они не различаются согласно определению 4.2.4.

Очевидно, что построение отношений эквивалентности на множестве состояний и на множестве магазинных символов должно выполняться совместно.

Эквивалентность входных символов прямого просмотра устанавливается после того, как в его управляющей таблице произведен переход к классам эквивалентных состояний и магазинных символов.

Определение **4.2.6.** Два входных символа прямого просмотра $a, b \in \Sigma^f$ считаются эквивалентными, если в каждом состоянии, в котором они допустимы, множества резольверных входов для них равны, а соответствующие управляющие элементы одинаковы, т.е. если для всех $q \in Q^f$ выполняются равенства $\delta_1^f(q, a) = \delta_1^f(q, b)$ и для всех $r \in \delta_1^f(q, a)$ (или, что то же самое, $r \in \delta_1^f(q, b)$) имеют место равенства $\delta_2^f(q, a, r) = \delta_2^f(q, b, r)$. Здесь δ_1^f и δ_2^f — компоненты управляющей таблицы прямого просмотра, в которых произведен переход к классам эквивалентных состояний и магазинных символов.

Эквивалентность входных символов обратного просмотра, в качестве которых, как известно, используются состояния прямого просмотра, определяется аналогично эквивалентности входных символов прямого просмотра.

4.3. СОКРАЩЕНИЕ ЧИСЛА ВХОДОВ В ТАБЛИЦУ ВОЗВРАТНЫХ СОСТОЯНИЙ

Дополнительная оптимизация прямого просмотра возможна за счет сокращения числа входов по магазинным символам в таблице возвратных состояний. Именно, если после перехода в таблице возвратных состояний к классам эквивалентности магазинных символов и состояний обнаруживается, что для некоторого класса магазинных символов класс эквивалентности возвратных состояний не зависит от того, какому классу подавляемых состояний он соответствует, то такой класс магазинных символов исключается из входа в таблицу возвратных состояний. Точнее, если при фиксированных $r \in 2^{\Re^{f*}}$ и $X \in \Gamma^f$ для всех $q \in Q^f$, для которых $\delta_3^f(q, X, r)$ определено, $\delta_3^f(q, X, r)$ имеет одно и то же значение, равное, скажем, $p \in Q^f$, то магазинный символ X во всех магазинных цепочках должен быть заменен на состояние p и исключен из входов в таблицу возвратных состояний δ_3^f. Во время процессирования такой магазинный символ, будучи снят с вершины магазина, используется непосредственно как возвратное состояние (без обращения к таблице возвратных состояний).

4.4. ПОРЯДОК ОПТИМИЗАЦИОННЫХ ПРЕОБРАЗОВАНИЙ

Оптимизация управляющих таблиц челночного процессора производится в следующем порядке:

1. Построить отношение эквивалентности R_1 на множестве состояний обратного просмотра (безотносительно к прямому) и соответствующие классы эквивалентности C_1 в этом отношении эквивалентности.
2. В управляющей таблице обратного просмотра перейти к классам эквивалентности состояний C_1 (сокращение числа входов по состояниям в управляющую таблицу обратного просмотра).
3. Построить отношение эквивалентности R_2 на входных символах обратного просмотра, в качестве которых используются состояния прямого просмотра (без учета их эквивалентности как состояний прямого просмотра) и соответствующие классы эквивалентности C_2 в этом отношении эквивалентности.
4. Перейти в таблице обратного просмотра к классам эквивалентности C_2 (сокращение числа входов по входным символам обратного просмотра)[85].
5. Построить отношения эквивалентности на множестве состояний[86] R_3 и на множестве магазинных символов R_4 прямого просмотра, а также соответствующие им классы эквивалентности C_3 и C_4.
6. В магазинных цепочках прямого просмотра заменить магазинные символы на классы эквивалентности из C_4, которым они принадлежат.
7. В управляющей таблице прямого просмотра заменить переходные состояния на классы эквивалентности из C_3, которым они принадлежат (сокращение числа входов по состояниям в управляющую таблицу прямого просмотра).
8. В таблице возвратных состояний δ_3^f перейти к классам эквивалентности состояний из C_3 и магазинных символов из C_4, построенных на шаге 5 (сокращение таблицы возвратных состояний по этим двум входам).
9. В магазинных цепочках прямого просмотра каждый класс из C_4 заменить на класс возвратных состояний из C_3, если последний не зависит от классов подавляемых состояний, с которыми этот класс магазинных символов сопряжен. Попросту говоря, если таблица δ_3^f после ее преобразования на шаге 8 для некоторого класса магазинных символов $[X]$ для всех резольверных входов дает один и тот же класс возвратных состояний $[q]$ по всем классам подавляемых состояний, с которыми этот класс магазинных символов $[X]$ сопряжен, то во всех магазинных цепочках управляющих элементов магазинный класс $[X]$ должен быть заменен на класс состояний $[q]$.

---

[85] Соответственно обратный просмотр, получив из входного потока номер класса состояний прямого просмотра в качестве своего входного символа, обращается к управляющей таблице обратного просмотра через лексический переходник обратного просмотра, дающий номер соответствующего входа по классам входных символов обратного просмотра. Способ построения такого переходника обратного просмотра описывается в разд. 4.5.

[86] С учетом их эквивалентности как входных символов обратного просмотра.

10. Построить отношения эквивалентности R_5 на множестве входных символов прямого просмотра и соответствующие классы эквивалентности C_5 в этом отношении.
11. Перейти в управляющей таблице прямого просмотра к классам эквивалентности входных символов из C_5 (сокращение числа входов по входным символам в управляющей таблице прямого просмотра)[87].

4.5. ПОСТРОЕНИЕ ЛЕКСИЧЕСКИХ ПЕРЕХОДНИКОВ ПРЯМОГО И ОБРАТНОГО ПРОСМОТРОВ

Лексический переходник прямого просмотра — это таблица, которая по первоначальному лексическому классу входной лексемы, поставляемой сканером, выдает номер входа по входным символам в управляющую таблицу прямого просмотра.

Для построения такого переходника достаточно сначала построить список номеров $1, 2, \ldots, n$, где n — число первоначальных лексических классов, а затем каждый элемент этого списка заменить на номер соответствующего класса эквивалентности из C_5, которому принадлежит этот элемент. Точнее, если некоторый входной символ сканер относит к первоначальному лексическому классу номер i, то этот номер i в упомянутой последовательности заменяется на номер k, где k — номер класса эквивалентности, к которому этот символ отнесен оптимизатором таблиц.

Лексический переходник прямого просмотра строится оптимизатором таблиц автоматически.

Лексический переходник обратного просмотра — это таблица, которая по номеру класса эквивалентных состояний прямого просмотра определяет номер входа по входным символам в управляющую таблицу обратного просмотра.

Способ построения лексического переходника обратного просмотра так же прост. Составляется последовательность номеров: $1, 2, \ldots, m$, где m — число классов эквивалентности состояний прямого просмотра. В этой последовательности номер i заменяется на k, если любое из состояний, входящих в класс эквивалентных состояний прямого просмотра номер i, содержится в классе эквивалентных входных символов обратного просмотра номер k. Лексический переходник обратного просмотра строится оптимизатором таблиц автоматически.

[87] Соответственно прямой просмотр использует лексический переходник, который определяет номер входа для каждого входного символа. Способ построения такого переходника описывается в разд. 4.5.

4.6. ИЛЛЮСТРАЦИЯ МЕТОДА ОПТИМИЗАЦИИ ПРОЦЕССОРА

Оптимизация калькулятора. Рассмотрим метод оптимизации процессоров на примере калькулятора арифметических выражений, обсуждавшегося в гл. 1. В результате его применения к сплайновому процессору, моделирующему калькулятор, получаем следующие классы эквивалентности состояний, магазинных и входных символов прямого просмотра[88]:

1. Классы эквивалентности состояний

1: {1} 6: {7} 11: {17}
2: {2} 7: {12} 12: {18}
3: {3 4 5 8 9 10 11} 8: {13} 13: {19}
4: {5} 9: {14 15}
5: {6} 10: {16}

2. Классы эквивалентности магазинных символов

1: {1}
2: {2}

3. Классы эквивалентности входных символов

1: {d} 4: {+ –} 7: {)}
2: {.} 5: {* /} 8: {EOF}
3: {E} 6: {(}

Замена состояний, магазинных и входных символов на соответствующие классы эквивалентности, которым они принадлежат, с учетом того, что возвратные состояния не зависят от подавляемых, дает оптимизированную управляющую таблицу калькулятора — см. табл. 4.1. Она также состоит из маленьких табличек, в заголовке каждой из них указан состав класса эквивалентных состояний, к которому она относится.

Таблица 4.1

Вход	Семантика	Магазин	Состояние
1	**2**	**3**	**4**
	Класс состояний 1={1}		
[d]	Reset;InitInt;SetDig	-10	2
[+]	Reset;PushMonOp	-10	3
[(]	Reset;PushOpenPar	-10	4

[88] Обратный просмотр в калькуляторе не используется.

Продолжение табл. 4.1

1	2	3	4
	Класс состояний 2={2}		
ε	AppDig; SetIntPart; PushOpd		**Sup**
[d]	AppDig;SetDig		2
[.]	AppDig;SetIntPart		5
[E]	AppDig;SetIntPart		6
[+]	AppDig; SetIntPart; PushOpd; PushDyadicOp		3
[*]	AppDig; SetIntPart; PushOpd; PushDyadicOp		3
	Класс состояний 3={3 4 8 9 10 11}		
[d]	InitInt;SetDig		2
[+]	PushMonOp		3
[(]	PushOpenPar		4
	Класс состояний 4={5}		
[d]	InitInt;SetDig	-11	2
[+]	PushMonOp	-11	3
[(]	PushOpenPar	-11	4
	Класс состояний 5={6}		
[d]	InitInt;SetDig		7
	Класс состояний 6={7}		
[d]	SetSign;InitInt;SetDig		8
[+]	SetSign		9
	Класс состояний 7={12}		
ε	AppDig;SetFrPart;AppFrPart;PushOpd		**Sup**
[d]	AppDig;SetDig		7
[E]	AppDig;SetFrPart;AppFrPart		6
[+]	AppDig;SetFrPart;AppFrPart;PushOpd;PushDyadicOp		3
[*]	AppDig;SetFrPart;AppFrPart;PushOpd;PushDyadicOp		3
	Класс состояний 8={13}		
ε	AppDig;SetExp;AppExpPart;PushOpd		**Sup**
[d]	AppDig;SetDig		8
[+]	AppDig;SetExp;AppExpPart;PushOpd;PushDyadicOp		3
[*]	AppDig;SetExp;AppExpPart;PushOpd;PushDyadicOp		3
	Класс состояний 9={14 15}		
[d]	InitInt;SetDig		8
	Класс состояний 10 ={16}		
[EOF]	Complete		1 2

Окончание табл. 4.1

1	2	3	4
	Класс состояний 11 ={17}		
)	UnloadAndDiscardOpenPar		13
	Класс состояний 12 ={18} (конечный)		
ε			**Stop**
	Класс состояний 13 ={19}		
ε			**Sup**
[+]	PushDyadicOp		3
[*]	PushDyadycOp		2

В графе Вход перечислены допустимые классы эквивалентности входных символов. Каждый такой класс представлен первым своим элементом, заключенным в прямоугольные скобки.

Графа Семантика, как обычно, содержит цепочки семантических символов, которые следует интерпретировать в данном состоянии управления.

Графа Магазин задает магазинные цепочки, в которых отрицательными номерами представлены магазинные символы — возвратные состояния.

Графа Состояние задает переходный класс состояний или содержит символ **Sup** — указание на ε-движение при котором, как известно, используется верхний символ магазина: если он представлен положительным числом, то происходит обычное использование оптимизированной таблицы возвратных состояний; если он представлен отрицательным номером, то это номер возвратного состояния со знаком минус, который и используется в качестве следующего состояния управления.

Далее мы приведем размеры компонент управляющей таблицы калькулятора до (табл.4.2) и после (табл.4.3) оптимизации.

Размеры компонент в этих таблицах даны в элементах и в байтах. Например, компонента "Характеристики состояний : Descr" состоит из 19 элементов — дескрипторов, занимающих в памяти 76 байтов, а компонента "Характеристики состояний : Pool" состоит из 19 элементов — номеров вершин управляющей граф-схемы, занимающих в памяти 38 байтов.

Следует заметить, что приведенные оценки несколько расходятся с реальными цифрами, так как объекты, используемые для представления соответствующих компонент таблиц, содержат дополнительные поля данных, не показанные в этих таблицах.

З*амечание* **1.** Более точные цифры, относящиеся к длине двоичных файлов, содержащих соответствующие варианты управляющих таблиц калькулятора, равны соответственно 1079 и 845 байтов, а показатель оптимизации составляет 78%.

З*амечание* **2.** В результате оптимизации исключена таблица возвратных состояний. Этого, впрочем, и следовало ожидать, так как оригинальная таблица возвратных состояний калькулятора (см. табл. 1.2) демонстрирует независимость возвратных состояний от подавляемых. Возвратные состояния в оптимизированной управляющей таблице появляются лишь как элементы магазинных цепочек, представленные отрицательными номерами[89].

Таблица 4.2

КОМПОНЕНТА	Размер компоненты	
	в элементах	в байтах
Характеристики состояний: Descr	19	76
Pool	19	38
Лексический вход: Descr	19	76
Pool	46	276
Резольверный вход	0	0
Пул управляющих элементов	46	276
Семантические цепочки: Descr	20	80
Pool	39	78
Резольверные цепочки: Descr	0	0
Pool	0	0
Магазинные цепочки: Descr	2	8
Pool	2	4
Магазинные символы: Descr	2	8
Pool	2	4
Конечные состояния	1	2
Таблица возвр. состояний: Descr	1	4
Pool	2	8
	Всего:	938

[89] Для отличия их от обычных магазинных символов, которые представляются положительными номерами.

Таблица 4.3

КОМПОНЕНТА	Размер компоненты	
	в элементах	в байтах
Характеристики состояний: Descr	13	52
Pool	19	38
Лексический вход: Descr	13	52
Pool	34	204
Резольверный вход	0	0
Пул управляющих элементов	29	174
Семантические цепочки: Descr	20	80
Pool	39	78
Резольверные цепочки: Descr	0	0
Pool	0	0
Магазинные цепочки: Descr	2	8
Pool	2	4
Магазинные символы: Descr	2	8
Pool	2	4
Конечные состояния	1	2
Таблица возвр. состояний: Descr	0	0
Pool	0	0
	Всего:	704

В заключение этого раздела приведем оптимизированные управляющие таблицы для рекурсивного варианта[90] вычисления n! (табл. 4.4) и функции Аккермана (табл. 4.5). В обоих случаях в результате оптимизации таблицы возвратных состояний исключаются.

Таблица 4.4

Условие	Действия	Магазин	Состояние
1	**2**	**3**	**4**
Класс состояний 1 = {1}			
	Start		2
Класс состояний 2 = {2}			
Valid	Open_F_in_S		3
Default	Alarm		4
Класс состояний 3 = {3}			
Large	Open_F_in_F	-7	5
Small	SetOne	-7	6

[90] В примере 1.4 итеративного варианта вычисления факториала (см. разд. 1.6) приведена уже оптимизированная управляющая таблица.

Таблица 4.4 (продложение)

1	**2**	**3**	**4**
Класс состояний 4 = {4 11} (конечное)			
			Stop
Класс состояний 5 = {5}			
Large	Open_F_in_F	-8	5
Small	SetOne	-8	6
Класс состояний 6 = {6 10}			
			Sup
Класс состояний 7= {7}			
	Close_F_in_S		9
Класс состояний 8 = {8}			
	Close_F_in_F		6
Класс состояний 9 = {9}			
	Finish		4

Таблица 4.5

Условие	Действие	Магазин	Состояние
1	**2**	**3**	**4**
Класс состояний 1 = {1}			
	Start		2
Класс состояний 2 = {2}			
Valid	Open_A_in_Main		4
Default			3
Класс состояний 3 = {3}			
	Finish		5
Класс состояний 4 = {4}			
Condition1	SetResult	-9	6
Condition2	Open1_A_in_A	-9	7
Condition3	Open2_A_in_A	-9	8
Класс состояний 5 = {5 12 } (конечный)			
			Stop
Класс состояний 6 = {6 13 17}			
			Sup

Таблица 4.5 (продложение)

1	2	3	4
Класс состояний 7= {7 15}			
Condition1	SetResult	-10	6
Condition2	Open1_A_in_A	-10	7
Condition3	Open2_A_in_A	-10	8
Класс состояний 8 = {8}			
Condition1	SetResult	-11	6
Condition2	Open1_A_in_A	-11	7
Condition3	Open2_A_in_A	-11	8
Класс состояний 9 = {9}			
	Close_A_in_Main		5
Класс состояний 10 = {10 16}			
	Close_A_in_A		6
Класс состояний 11 = {11}			
	Close_A_in_A		12
Класс состояний 12 = {14}			
	Open3_A_in_A		7

4.7. УЧЕТ ДИАГНОСТИЧЕСКИХ СООБЩЕНИЙ ПРИ ОПТИМИЗАЦИИ

Оптимизация управления несколько ухудшает точность диагностики ошибок. Это связано с тем, что используемое отношение эквивалентности на множестве состояний не принимает в расчет еще один возможный различительный признак — диагностики. При такой оптимизации ничего не остается, как объединять диагностики, относящиеся ко всем состояниям одного класса, из-за чего и ухудшается точность характеризации ошибок. Альтернативное определение отношения эквивалентности состояний, учитывающее диагностики как еще один различительный признак, не было использовано, так как платой за точность диагностики ошибок было бы увеличение объема управляющих таблиц. Из двух возможностей — точные диагностики при больших управляющих таблицах или чуть менее точные диагностики при меньших по объему таблицах — была выбрана вторая.

Заметим, что учет диагностик при определении эквивалентности состояний потребовал бы частичного изменения технологической последовательности: оптимизации управляющих таблиц должна была бы выполняться *после* генерации диагностических сообщений.

4.8. ОПТИМИЗАЦИЯ УПРАВЛЯЮЩИХ ГРАФ-СХЕМ

Оптимизация управляющих граф-схем реализуется на основе информации, имеющейся в оптимизированной управляющей таблице. Действительно, сравнивая характеристики эквивалентных состояний между собой можно определить, какие вершины управляющей граф-схемы могут быть "склеены". Выполнив склейку таких вершин, при которой склеятся также и соответствующие дуги, мы получим управляющую граф-схему с меньшим числом вершин и дуг.

Редуцированная управляющая граф-схема может быть полезна, если она используется для непосредственного управления работой челночного процессора, а также при генерации тестов.

Заметим, что несмотря на то, что любая RBNF-грамматика отображается в граф-схему, не всякая граф-схема, в частности, полученная в результате оптимизации, может оказаться отображаемой в форму RBNF-грамматики *адекватно* в том смысле, что не существует такой RBNF-грамматики, которая отображалась бы в эту граф-схему. Другими словами, это отображение не является биекцией.

Редуцированная грамматика, если она существует, получается попутно в технологическом потоке. Она представляет результат тех эквивалентных преобразований исходной грамматики, которые иначе было бы необходимо выполнять вручную или на компьютере с помощью отдельной техники, например, на базе системы Рефал.

Редуцированная грамматика адекватная минимизированной граф-схеме позволяет увидеть явно, по какой грамматике был построен оптимизированный языковый процессор. В этом случае редуцированную грамматику можно использовать для построения процессора, и он будет сразу оптимальным без какой бы то ни было оптимизации.

Глава 5

ЭКВИВАЛЕНТНЫЕ ПРЕОБРАЗОВАНИЯ ТРАНСЛЯЦИОННЫХ ГРАММАТИК

5.1. ПРИЧИНЫ, ЦЕЛИ И МЕТОДЫ ПРЕОБРАЗОВАНИЙ

Ограничения механизма анализа. Из метода построения сплайновых процессоров, описанных в гл. 3, следует, что трансляционная грамматика должна удовлетворять определенным ограничениям, чтобы по ней можно было построить адекватный сплайновый процессор. Как уже упоминалось, эти ограничения вытекают из требования детерминированности механизма реализации трансляции и состоят в следующем:

1. Разложение любого состояния прямого просмотра должно быть конечно.

Как определено в гл. 3, разложение состояния прямого просмотра представляется в виде конечного множества деревьев (леса). Ограничение 1 означает, что все ветви этих деревьев должны иметь конечную длину. По существу это равносильно недопустимости левосторонней рекурсии в правилах грамматики.

2. Разложение каждого состояния прямого просмотра должно быть сбалансировано по одноименным терминальным листьям.

Это значит, что все ветви разложения, листья которых помечены одинаковыми терминальными символами грамматики, относящиеся к одному и тому же контексту[91], должны иметь одну и ту же длину. Выполнение этого требования гарантирует однозначное отнесение каждого входного символа к определенному уровню конструкций входного языка и, следовательно, детерминизм манипуляций с магазином управляющего процессора. Именно, однозначно определено, сколько и каких магазинных символов надо записать в магазин при приеме каждого входного символа.

3. Разложение каждого состояния прямого просмотра должно быть сбалансировано по конечным листьям.

Это значит, что все ветви разложения, листья которых помечены метками вида "end-*A*", где *A* — нетерминалы (не обязательно одинаковые), относящиеся к одному и тому же контексту, должны иметь одинаковую длину. Выполнение этого требования гарантирует однозначное определение, сколько и каких магазинных символов следует записать в магазин перед тем, как использовать верхний символ магазина для определения возвратного состояния.

4. Разложение каждого состояния прямого просмотра должно быть семантически сбалансировано.

[91] Считается, что ветви разложения состояния относятся к одному и тому же контексту, если цепочки резольверных символов прямого просмотра, их помечающие, одинаковы.

Это значит, что все ветви разложения, о которых шла речь в двух предыдущих пунктах, должны помечаться одинаковыми семантическими цепочками, если в них принимать во внимание только семантические символы прямого просмотра.

Ограничения 1–4 определяют внутренние свойства разложения каждого состояния прямого просмотра. Последнее требование является внешним по отношению к состояниям.

5. Каждый входной символ, принимаемый в подавляемом состоянии, не должен приниматься ни в каком из возвратных состояний, производном от этого состояния и сопряженных с ним магазинных символов.

Ограничение 5 надо воспринимать "рекурсивно". Нарушение этого ограничения означало бы неопределенность в действиях прямого просмотра — принимать ли текущий входной символ в текущем состоянии (считать текущий входной символ принадлежащим текущей конструкции) или его следует анализировать в возвратном состоянии (т.е. отнести его к внешней конструкции).

Таким образом, необходимость или желательность эквивалентных преобразований трансляционной грамматики, специфицирующей некоторую трансляцию, может возникнуть по двум основным причинам:

управляющая грамматика не принадлежит классу грамматик, к которому применим метод автоматического синтеза языкового процессора типа, используемого в технологическом комплексе SYNTAX; необходимо построить эквивалентную грамматику требуемого класса;

желательно упростить синтаксическую структуру входного языка за счет достижения большей степени регулярности грамматики; практически это упрощение сводится к сокращению числа нетерминалов и правил грамматики для них и соответственно к снижению интенсивности использования магазина процессора; в предельном случае в грамматике может остаться только один (начальный) нетерминал и единственное RBNF-правило для него, и тогда[92] соответствующий процессор, реализующий трансляцию, будет не сплайновым, а конечным.

Часто, преследование второй из этих двух целей приводит одновременно и к достижению первой.

Методы эквивалентных преобразований. Из определения трансляционной грамматики как совокупности управляющей грамматики и описания операционной среды, включающей интерпретацию семантических и резольверных символов, следует, что для ее эквивалентных преобразований существуют две возможности: 1) преобразовывать управляющую грамматику, заботясь о сохранении определяемого ею синтаксического управления (такие преобразования мы будем называть *синтаксически эквивалентными*), либо 2) преобразовывать

[92] При условии, конечно, что начальный нетерминал не встречается в правой части этого RBNF-правила грамматики.

интерпретацию семантических и резольверных символов, быть может даже меняя их набор и элементы операционной среды, следя лишь за сохранением самой трансляции, т. е. результирующего состояния операционной среды для всех входных цепочек (такие преобразования мы будем называть *семантически эквивалентными*; очевидно, что любое синтаксически эквивалентное преобразование является также и семантически эквивалентным)[93].

На практике используются обе возможности. Поскольку, как уже отмечалось, исключение нетерминалов из правил благотворно сказывается не только на эффективности процессора, но и на самой возможности его построения, то начнем обсуждение с синтаксически эквивалентных преобразований, позволяющих исключить все несамовставленные нетерминалы. Напомним, что нетерминал $A \in N$ называется *самовставленным*, если существует вывод вида:

$A \overset{*}{\Rightarrow} \alpha A \beta$, где α и $\beta \in W^{+}$, $W = N \cup T \cup \Re \cup \Sigma$.

Другими словами, нетерминал A — самовставленный, если он выводится сам из себя в одновременно непустых левом (α) и правом (β) контекстах. Если нетерминал не является самовставленным, то он *несамовставленный*. В последнем случае он не может быть одновременно и лево- и право- рекурсивным, т. е. таким, что одновременно существуют выводы вида:

$A \overset{*}{\Rightarrow} \alpha A$ (праворекурсивность) и $A \overset{*}{\Rightarrow} A \beta$ (леворекурсивность).

И в самом деле, в противном случае из двух этих выводов можно было бы построить вывод вида $A \overset{*}{\Rightarrow} \alpha A \beta$.

Из теории формальных грамматик известно[94], что язык, порождаемый КС-грамматикой без самовставлений, является *регулярным*, т.е. распознаваемым некоторым конечным автоматом[95]. По следствию из теоремы Клини в этом случае существует регулярное выражение над терминалами грамматики, представляющее тот же самый язык. Оно может быть получено посредством эквивалентных преобразований ее правил.

Аналогичные синтаксически эквивалентные преобразования можно применять и к управляющим КС-грамматикам без самовставлений с целью получить регулярное выражение над терминалами, резольверами и семантиками, специифицирующее то же самое синтаксическое управление. И тогда по этому регулярному выражению можно построить конечный процессор, реализующий ту же самую трансляцию. Во всяком случае, исключение из грамматики несамовставленных нетерминалов всегда возможно.

Далее описывается алгоритм решения этой задачи.

---

[93] Синтаксически эквивалентные преобразования обсуждаются и иллюстрируются далее в этой главе. Пример совместного использования синтаксических и семантических преобразований приводится в гл. 6.

[94] См., например, [2].

[95] Однако из этого не следует, что грамматика с самовставленными нетерминалами не может порождать регулярный язык.

Алгоритм 5.1. Исключение несамовставленных нетерминалов из управляющей КС-грамматики.

Вход: $G_C = (N, T, \Re, \Sigma, P, S)$ — приведенная управляющая BNF-грамматика.

Выход: $G_C' = (N', T', \Re', \Sigma', P', S')$ — управляющая RBNF-грамматика, не содержащая несамовставленных нетерминалов в правых частях правил, синтаксически эквивалентная грамматике G_C.

Метод:

1. Разобьем алфавит нетерминалов N данной грамматики G_C на два непересекающихся подмножества N_1 и N_2, включив в N_1 все несамовставленные нетерминалы, а в N_2 — все остальные (самовставленные): $N = N_1 \cup N_2$[96].

2. Упорядочим каким-либо образом нетерминалы из N_1. Пусть, например, $N_1 = \{A_1, A_2, \ldots, A_m\}$.

3. Рассмотрим правило для нетерминала $A_1 \in N_1$. В общем случае оно имеет вид

$$A_1 ::= A_1\alpha_1 \mid A_1\alpha_2 \mid \ldots \mid A_1\alpha_n \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_k \qquad (5.1)$$

или

$$A_1 ::= \alpha_1 A_1 \mid \alpha_2 A_1 \mid \ldots \mid \alpha_n A_1 \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_k, \qquad (5.2)$$

где в первую очередь перечислены все явнорекурсивные альтернативы, в которых $\alpha_i, \beta_j \in (N \cup \Sigma \cup \Re)^*$ $(i = 1, 2, \ldots, n;\ j = 1, 2, \ldots, k)$ не содержат нетерминала A_1.

Очевидно, что правила (5.1), (5.2) эквивалентны правилам вида соответственно

$$A_1 : (\beta_1 ; \beta_2 ; \ldots ; \beta_k), (\alpha_1 ; \alpha_2 ; \ldots ; \alpha_n)^*, \qquad (5.1')$$

$$A_1 : (\alpha_1 ; \alpha_2 ; \ldots ; \alpha_n)^*, (\beta_1 ; \beta_2 ; \ldots ; \beta_k). \qquad (5.2')$$

4. (Прямой ход). Пусть правила для первых p нетерминалов из N_1 уже приведены к виду

$$A_i : \varphi_i, \psi_i^*. \text{ или } A_i : \psi_i^*, \varphi_i. \qquad (5.3)$$

где φ_i, ψ_i — некоторые регулярные выражения над символами из W, не содержащие нетерминалов $A_1, A_2, \ldots, A_i$ (i =1,2, ..., p).

Преобразуем правило для A_{p+1} к виду (5.3). Для этого в правую часть правила для A_{p+1} вместо каждого вхождения A_j $(j = 1, 2, \ldots, p)$ последовательно[97] подставим правую часть определяющего его правила; таким образом из правой части правила для A_{p+1} будут исключены нетерминалы $A_1, A_2, \ldots, A_p$, причем если A_{p+1} входит в полученное регулярное выражение, то только в одном из двух соответствующих контекстов[98]

---

[96] Существует очевидный алгоритм, позволяющий определить, является ли данный нетерминал самовставленным или нет.

[97] Т.е. сначала выполняются подстановки при $j = 1$, затем полученные выражения преобразуются подстановками при $j = 2$, и т.д.

[98] В противном случае вопреки предположениям какие-то из A_i $(i = 1, 2, \ldots, p)$ оказались бы самовставленными.

$A_{p+1} : A_{p+1}, \psi_{p+1}; \varphi_{p+1}.$ или $A_{p+1} : \psi_{p+1}, A_{p+1} ; \varphi_{p+1}.$,

где φ_{p+1} и ψ_{p+1} — регулярные выражения, не содержащие нетерминалов $A_1, A_2, \ldots, A_{p+1}$; после этого заменим крайние рекурсии на итерации и правило для A_{p+1} окончательно преобразуем соответственно к виду $A_{p+1} : \varphi_{p+1}, \psi_{p+1}{}^*$. или $A_{p+1} : \psi_{p+1}{}^*, \varphi_{p+1}$.

5. Повторяя шаг 4 для $p = 1, 2, \ldots, m-1$ получаем правила требуемого вида для всех несамовставленных нетерминалов из N_1.

Заметим, что полученное таким образом правило для A_m в своей правой части не содержит ни одного несамовставленного нетерминала.

6. (Обратный ход). Для каждого $p = m-1, m-2, \ldots, 1$, подставляя в правую часть правила, определяющего A_p, выражения, полученные для A_m, A_{m-1},..., A_{p+1}, окончательно исключим несамовставленные нетерминалы из правых частей правил для несамовставленных нетерминалов.

7. Положим $N' = \{S\} \cup N_2$, $T' = T$, $\Re' = \Re$, $\Sigma' = \Sigma$, $S' = S$.

8. Наконец, преобразуем правила для нетерминалов из N', подставив в их правые части выражения для несамовставленных нетерминалов, полученные на предыдущих шагах алгоритма. В результате будут получены искомые правила P', определяющие нетерминалы из N', которые не содержат несамовставленных нетерминалов вовсе.

Очевидно, если $N_2 = \varnothing$, то грамматика G_C' — явнорегулярна: в ней имеется единственное правило для нетерминала S, правая часть которого есть регулярное выражение над терминальными, резольверными и семантическими символами.

***Замечание* 1.** Предложенный алгоритм в некотором смысле аналогичен методу Гаусса для решения систем линейных алгебраических уравнений, в котором сначала (на прямом ходу) система приводится к “треугольному” виду, а затем (на обратном ходу) посредством последовательных подстановок находятся значения неизвестных.

***Замечание* 2.** Если данная управляющая грамматика без самовставлений, то применение к ней предложенного алгоритма дает явнорегулярную управляющую грамматику, синтаксически эквивалентную исходной.

***Замечание* 3.** Принимая во внимание, что "сверхзадачей" алгоритма является получение для начального нетерминала грамматики регулярного выражения, не зависящего от несамовставленных нетерминалов, можно исключить из него шаг 6 путем специального упорядочивания нетерминалов грамматики таким образом, чтобы начальный нетерминал оказался последним.

***Замечание* 4.** Несмотря на то, что описанный здесь алгоритм аналогичен алгоритму 2.1 из монографии [2], он применим к более общему классу систем правил грамматик, нежели стандартные линейные системы с регулярными коэффициентами.

5.2. НЕКОТОРЫЕ ПОЛЕЗНЫЕ РЕГУЛЯРНЫЕ ТОЖДЕСТВА

При синтаксически эквивалентных преобразованиях могут быть полезны следующие регулярные тождества, в которых A, B и C — произвольные регулярные выражения[99]:

$$A;B=B;A \tag{5.4}$$
$$(A;B);C=A;(B;C) \tag{5.5}$$
$$(A,B),C=A,(B,C) \tag{5.6}$$
$$(A;B),C=A,C;B,C \tag{5.7}$$
$$C,(A;B)=C,A;C,B \tag{5.8}$$
$$A,\varepsilon=\varepsilon,A=A \tag{5.9}$$
$$A,A^*=A^*,A=A^+ \tag{5.10}$$
$$(A,B)^*,A=A\#B \tag{5.11}$$
$$(A\#B)\#C=A\#(B;C) \tag{5.12}$$
$$A\#(B\#C)=A,(\varepsilon;B\#(A;C),A) \tag{5.13}$$
$$(A^+)\#B=A\#(\varepsilon;B) \tag{5.14}$$
$$[A]=A;\varepsilon \tag{5.15}$$

5.3. ИЛЛЮСТРАЦИЯ АЛГОРИТМА ИСКЛЮЧЕНИЯ НЕТЕРМИНАЛОВ

Продемонстрируем описанный алгоритм на примере грамматики, определяющей числа в языке программирования Алгол 68. Заодно это послужит свидетельством возможности применения SYNTAX-технологии к языкам, определяемым альтернативными средствами.

Из [1] имеем следующее определение чисел посредством грамматики А. ван Вейнгаардена:

NUMERAL :: fixed point numeral; (1.1)
variable point numeral; (1.2)
floating point numeral. (1.3)
fixed point numeral : digit cypher sequence. (2)
digit cypher : DIGIT symbol. (3)
NOTION sequence : NOTION; NOTION, NOTION sequence. (4)
DIGIT:: digit zero; digit one; digit two; digit three; digit four;
digit five; digit six; digit seven; digit eight; digit nine. (5)
variable point numeral : integral part option, fractional part. (6)
integral part : fixed point numeral. (7)
fractional part : point symbol, fixed point numeral. (8)
floating point numeral : stagnant part, exponent part. (9)
stagnant part : fixed point numeral; (10.1)
variable point numeral. (10.2)

[99] Напомним, что ";" обозначает объединение, "," — конкатенацию, "#" — итерацию с разделителем, "$^+$" — усеченную итерацию, "*" — итерацию.

exponent part : times ten to the power choice, power of ten. (11)
times ten to the power choice : times ten to the power symbol; (12.1)
letter e symbol. (12.2)
power of ten : plusminus option, fixed point numeral. (13)
plusminus : plus symbol; minus symbol. (14)
NOTION option : NOTION; EMPTY. (15)
EMPTY :: . (16)

Дадим краткое определение грамматик А. ван Вейнгаардена и поясним, каким образом они используются для спецификации языков.

В отличие от обычных КС-грамматик, правила грамматики А. ван Вейнгаардена являются *двухуровневыми*. Один уровень представляют так называемые *гиперправила*, а другой — *метаправила*. Одно гиперправило фактически является схемой многих правил обычной КС-грамматики. Оно состоит из левой и правой частей, отделяемых друг от друга двоеточием ':'. В левой части находится *гиперпонятие* — некоторая последовательность *метапонятий* — цепочек больших синтаксических знаков (букв) и малых синтаксических знаков (букв), а в правой — некоторое множество *альтернатив*, отделенных друг от друга точкой с запятой ';'. Каждая альтернатива является конкатенацией некоторого числа *членов*, отделеляемых друг от друга запятыми ','. В качестве таких членов используются гиперпонятия и *понятия* — цепочки малых синтаксических знаков, для которых существуют определяющие их правила.

Метапонятия порождаются при помощи *метаправил*. Множество метаправил образует обычную КС-грамматику, в которой метапонятия играют роль нетерминалов, а малые синтаксические знаки — роль терминалов. Левая часть метаправила отделяется от правой двумя двоеточиями '::'. Из каждого метапонятия можно вывести множество *терминальных метапорождений* — цепочек малых синтаксических знаков, называемых *протопонятиями*.

Механизм получения правил грамматики из гиперправила прост: достаточно в гиперправиле каждое вхождение метапонятия заменить на какое-нибудь его терминальное метапорождение. При этом все вхождения одного и того же метапонятия должны быть заменены на одно и то же терминальное метапорождение. Такая замена называется *согласованной подстановкой*. В результате получается правило, в котором используются только цепочки, составленные из малых синтаксических знаков. Если такая цепочка заканчивается на 'symbol', то она представляет терминал (символ), в противном случае — нетерминал (понятие). Для каждого терминала определяется один или несколько способов конкретного представления. Например, для терминала 'plus symbol' дается конкретное представление '+'.

Полученные таким образом правила КС-грамматики могут использоваться для вывода предложений языка обычным образом[100].

[100] Строго говоря, такую грамматику нельзя считать КС-грамматикой, так как в общем случае она может содержать бесконечное множество нетерминалов и бесконечное множество правил.

В рассматриваемом примере NUMERAL, DIGIT, NOTION, EMPTY — метапонятия, а (1.1)–(1.3), (5) и (16) — метаправила. Гиперправила — (3), (4) и (15), остальное — правила.

Гиперправило (3) равносильно правилу

digit cypher : digit zero symbol ; digit one symbol ;
digit two symbol ; digit three symbol ;
digit four symbol ; digit five symbol ;
digit six symbol ; digit seven symbol ;
digit eight symbol ; digit nine symbol.

которое получается путем подстановки в (3) всех возможных терминальных метапорождений метапонятия DIGIT.

Чтобы получить правило для понятия digit cypher sequence, используемого в правой части правила (2), достаточно в гиперправило (4) вместо всех вхождений метапонятия NOTION, подставить digit cypher в качестве терминального метапорождения NOTION. В результате получаем правило

digit cypher sequence : digit cypher ;
digit cypher sequence, digit cypher.

Чтобы получить правило для понятия integral part option, используемого в правой части правила (6), достаточно в гиперправило (16) вместо метапонятия NOTION подставить его терминальное метапорождение integral part. Это даст правило

integral part option : integral part ; .

Таким образом, мы получаем следующую обычную КС-грамматику, определяющую числа в Алголе 68[101], в которой терминальные символы заменены их конкретными представлениями:

number : fixed point numeral ; (1.1)
variable point numeral ; (1.2)
floating point numeral. (1.3)
fixed point numeral : digit cypher sequence. (2)
digit cypher sequence : digit cypher ;
digit cypher sequence , digit cypher. (3)
digit cypher : '0'; '1'; '2'; '3'; '4'; '5'; '6'; '7'; '8'; '9'. (4)
variable point numeral : integral part option , fractional part. (5)
integral part option : integral part ; . (6)
integral part : fixed point numeral. (7)
fractional part : '.' , fixed point numeral. (8)
floating point numeral : stagnant part, exponent part. (9)
stagnant part : fixed point numeral ; variable point numeral. (10)
exponent part : times ten to the power choice , power of ten. (11)
times ten to the power choice : '\' ; 'e'. (12)

[101] Следует быть внимательным: пустые порождения не отмечены никаким заметным образом. Однако их можно увидеть "на фоне" соседних символов. См., например, правила (6) и (14) на следующей странице.

power of ten : plusminus option, fixed point numeral. (13)
plusminus option : plusminus ; . (14)
plusminus : '+' ; '–'. (15)

Заметим, что гиперправило (1.1)–(1.3) фактически играет роль заглавного правила в этой грамматике чисел. Чтобы оно выглядело как правило КС-грамматики, мы заменили в левой его части метапонятие NUMERAL понятием number, а метасинтаксическую связку ‘::’ разделителем ‘:’.

Начнем исполнение демонстрируемого алгоритма с констатации того факта, что все нетерминалы этой грамматики являются несамовставленными. Введем для них нумерующие краткие обозначения с учетом замечания, открывающего возможность обойтись без "обратного хода", т.е. без шага 6:

A_1 = plusminus
A_2 = plusminus option
A_3 = power of ten
A_4 = times ten to the power choice
A_5 = exponent part
A_6 = stagnant part
A_7 = floating point numeral
A_8 = fractional part
A_9 = integral part
A_{10} = integral part option
A_{11} = variable point numeral
A_{12} = digit cypher
A_{13} = digit cypher sequence
A_{14} = fixed point numeral
A_{15} = number

Начальным нетерминалом является number, и он — последний в заданном упорядочивании. В новых обозначениях правила приобретают более компактный вид:

A_{15} : A_{14} ; A_{11} ; A_7.
A_{14} : A_{13}.
A_{13} : A_{12} ; A_{13} , A_{12}.
A_{12} : '0' ; '1' ; '2' ; '3' ; '4' ; '5' ; '6' ; '7' ; '8' ; '9'.
A_{11} : A_{10} , A_8.
A_{10} : A_9 ; .
A_9 : A_{14}.
A_8 : '.' , A_{14}.
A_7 : A_6 , A_5.
A_6 : A_{14} ; A_{11}.
A_5 : A_4 , A_3.
A_4 : '\' ; 'e'.
A_3 : A_2 , A_{14}.
A_2 : A_1 ; .
A_1 : '+' ; '–'.

Сначала требуется путем подстановок и исключения левосторонней рекурсии привести правила к такому виду, при котором номера нетерминалов, используемых в правой части правила, были бы строго больше номера нетерминала в левой части правила.

Правило для A_1 уже имеет требуемый вид: в его правой части нет ни одного нетерминала.

В правиле для A_2 используется нетерминал A_1 с меньшим номером. Необходимо произвести подстановку выражения для A_1 в правую часть правила для A_2. Получаем:

A_2 : '+' ; '–' ; .

В правиле для A_3 справа используется A_2: требуется произвести подстановку выражения для A_2 в правило для A_3. Получаем:

A_3 : ('+' ; '–' ;) , A_{14}.

Теперь нетерминал в правой части (A_{14}) имеет номер, больший, чем нетерминал в левой части (A_3).

Правило для A_4 не нуждается в преобразованиях.

Правило для A_5 требуется преобразовать: вместо A_3 и A_4 подставим полученные для них выражения. Это дает

A_5 : ('\' ; 'e') , ('+' ; '–' ;) , A_{14}.

Правило для A_6 уже имеет требуемый вид.

Преобразуем правило для A_7, подставляя вместо A_6 и A_5 соответствующие выражения. Получаем:

A_7 : (A_{14} ; A_{11}) , A_4 , A_3.

Продолжая подстановки вместо A_3 и A_4, получаем:

A_7 : (A_{14} ; A_{11}) , ('\' ; 'e') , (('+' ; '–' ;) , A_{14}).

Правила для A_8 и A_9 не требуют преобразований, поскольку и так имеют требуемый вид.

Правило для A_{10} преобразуем посредством подстановки выражения для A_9. В результате получаем:

A_{10} : A_{14} ; . Короче, A_{10} : [A_{14}].

Аналогично правило для A_{11} преобразуется к виду

A_{11} : [A_{14}] , '.' , A_{14}.

Правило для A_{12} преобразовывать не требуется.

Правило для A_{13} приводится к виду

A_{13} : A_{12} ; A_{13} , A_{12} = A_{12}^{+} или A_{13} : d^{+}.

Здесь d обозначает любой символ цифры.

Соответственно правило для A_{14} приобретает вид

A_{14} : d^{+}.

Наконец, правило для A_{15} преобразуется следующим образом:

A_{15} : d^{+} ; [A_{14}] , '.' , A_{14} ; A_7 =

d^{+} ; [d^{+}], '.', d^{+} ; (A_{14} ; A_{11}), ('\' ; 'e'), (('+' ; '–' ;), A_{14}) =

d^{+} ; d^{*} , '.', d^{+} ; (d^{+} ; [A_{14}], '.', A_{14}), ('\'; 'e'), (('+' ; '–';), d^{+}) =

d^{+} ; d^{*} , '.' , d^{+} ; (d^{+} ; [d^{+}] , '.' , d^{+}), ('\'; 'e'), (('+' ; '–';), d^{+}) =

d^{+} ; d^{*} , '.', d^{+} ; (d^{+} ; d^{*} , '.', d^{+}), ('\'; 'e'), (['+' ; '–'], d^{+}) =

(d^{+} ; d^{*} , '.', d^{+}), [('\'; 'e'), (['+' ; '–'], d^{+})].

Таким образом, мы получили эквивалентную явнорегулярную грамматику с одним единственным правилом:

number : (d^{+} ; d^{*}, '.', d^{+}), [('\'; 'e'), (['+' ; '–'], d^{+})].

Здесь d = {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9'}. Как видим, "обратный ход" не потребовался.

В заключение этого параграфа перечислим основные синтаксически эквивалентные преобразования управляющих грамматик:

- вынесение общих членов альтернатив за скобки (направо или налево);
- подстановка вместо использующего вхождения нетерминала выражения, его определяющего;
- подстановка вместо некоторого подвыражения нового нетерминала, определяемого этим подвыражением (пополнение грамматики еще одним нетерминалом и соответствующим правилом для него);
- замена крайней (лево- или правосторонней) рекурсии на итерацию;
- замена обычной итерации, где возможно, итерацией с разделителем;
- использование тождеств, перечисленных в разд. 5.2, для упрощения вида правил.

5.4. РЕДУКЦИЯ КС-ГРАММАТИК

В этом параграфе мы обсудим использование информации об эквивалентностях состояний и входных символов, получаемой при оптимизации процессора прямого просмотра, для минимизации граф-схемы и отображения ее в редуцированную RBNF-грамматику, т. е. эквивалентную RBNF-грамматику с минимально возможным числом нетерминалов и терминалов. Проиллюстрируем применение этого алгоритма для приведения к минимальной форме КС-грамматики из предыдущего параграфа, порождающей регулярный язык чисел Алгола 68, и КС-грамматики простой модели арифметических выражений. Покажем, что задачу исключения несамовложенных нетерминалов из КС-грамматики можно решить, не прибегая к специальному алгоритму 5.1, использующем метод Гаусса, а полагаясь на механизм макроподстановок вспомогательных понятий при построении граф-схемы. Выясним, почему для минимальной граф-схемы, построенной по первой грамматики, не существует адекватная RBNF-грамматика. Опишем метод исключения всех нетерминалов из второй грамматики, несмотря на то, что все они, за исключением начального нетерминала, самовставленные.

Алгоритм 5.2. Минимизация граф-схемы по классам эквивалентности

Всюду, где в описании этого алгоритма говорится об отображении множества вершин граф-схемы в одну, надо понимать, что область этого преобразование не выходит рамки одной компоненты[102].

[102] Другими словами, “склеиваются” друг с другом только вершины, принадлежащие одной компоненте.

С учетом классов эквивалентных состояний и лексических классов входных символов, получаемых при оптимизации процессора, алгоритм минимизации граф-схемы состоит из следующих шагов:

1. Начиная с начального, рассматривается очередной класс эквивалентных состояний *C*.

(а) Если *C* — начальный класс, то он всегда состоит из одного начального состояния характеризуемого одной начальной вершиной граф-схемы. Эта вершина при преобразовании граф-схемы переходить сама в себя.

(б) Если *C* — класс не подавляемых состояний, то вершины этого множества состояний *V* всегда помечены терминалами из одного лексического класса *L*.

Преобразование граф-схемы: все вершины *V* отображаются в одну вершину *v*, помеченную лексическим классом *L*. Все дуги, инцидентные вершинам *V*, соединяются с *v* в качестве входных и выходных дуг соответственно.

(в) Если *C* — класс подавляемых состояний, то вершины *V* этого множества состояний могут быть помечены либо только терминалами, необязательно из одного лексического класса, либо только нетерминалами, не обязательно одинаковыми.

Преобразование граф-схемы: если вершины *V* — терминальные, то они разделяются на подмножества согласно лексическим классам, помечающих их терминальных символов. Все вершины одного подмножества преобразуются так же, как описано в п. (б). Если вершины *V* — нетерминальные, то они обрабатываются как в п. (г).

(г) Если *C* — класс возвратных состояний, то вершины этого множества состояний всегда помечены нетерминалами, причем необязательно различными.

Преобразование граф-схемы: все вершины *V*, помеченные одинаковыми нетерминалами, отображаются в одну вершину *v*, помеченную соответствующим нетерминалом. Все дуги, инцидентные вершинам-прообразам вершины *v*, соединяются с *v* в качестве входных и выходных дуг соответственно.

2. Процесс закачивается, когда рассмотрены все классы состояний.

Заметим, что случаи (в) и (г) относятся только к магазинным процессорам. Образы дуг наследуют метки дуг-прообразов.

Пример 5.1. Редукция грамматики чисел. Проиллюстрируем этот метод на примере грамматики чисел языка Алгол 68 (см. правила на стр. 130):

$A_{15} : A_{14} ; A_{11} ; A_7$.
$A_{14} : A_{13}$.
$A_{13} : A_{12} ; A_{13} , A_{12}$.
A_{12} : '0' ; '1' ; '2' ; '3' ; '4' ; '5' ; '6' ; '7' ; '8' ; '9'.
$A_{11} : A_{10} , A_8$.
$A_{10} : A_9 ; .$
$A_9 : A_{14}$.
A_8 : '.' , A_{14}.
$A_7 : A_6 , A_5$.
$A_6 : A_{14} ; A_{11}$.
$A_5 : A_4 , A_3$.
A_4 : '\' ; 'e'.
$A_3 : A_2 , A_{14}$.
$A_2 : A_1 ; .$
A_1 : '+' ; '–'.

Теперь порядок нетерминалов не важен, лишь бы нетерминал A_{15} оставался начальным. Если бы не леворекурсивное правило для A_{13}, то все нетерминалы, кроме A_{15}, можно было перенести из списка нетерминалов в список вспомогательных понятий и автоматически получить явнорегулярную грамматику. Воспользовавшись функцией Reduce ТК SYNTAX для исключения левой рекурсии из BNF-правил, мы получаем правило $A_{13} : A_{12}$, A_{12}^{*}[103] . Теперь ничто не мешает использовать механизм макроподстановок для получения явнорегулярной граф-схемы. Ее представление в виде диаграммы Вирта показано на рис. 5.1.

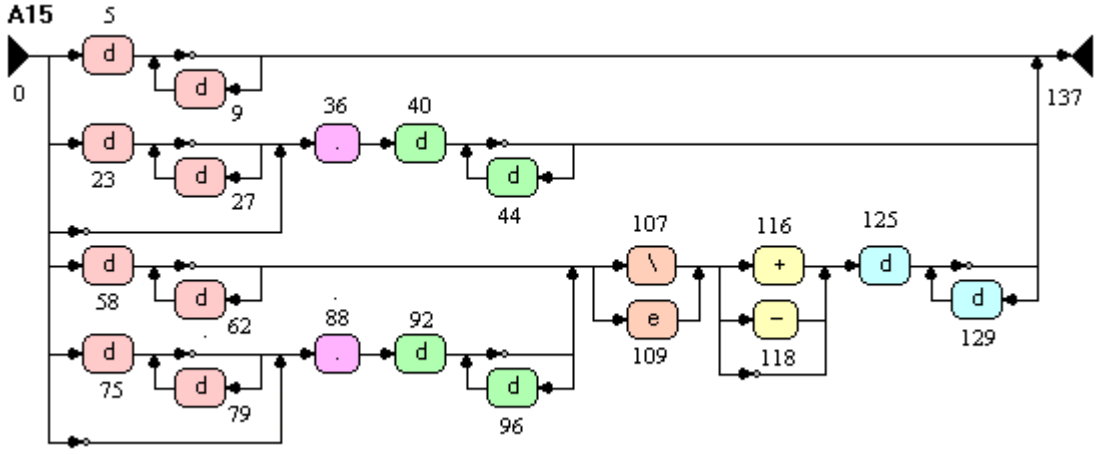


Рис. 5.1. Граф-схема A68NUM в виде диаграммы Вирта.

В ней при вершинах указаны номера записей, представляющих их во внутреннем представлении граф-схемы. Метки вершин, порождающих цифры, заменены на псевдотерминал ‘d’ только для того, чтобы не загромождать рисунок. Построенный по этой граф-схеме конечный процессор A68NUM имеет 12 состояний, характеристики которых приведены в табл. 5.1.

Таблица 5.1.

Сост.	Вершины	Сост.	Вершины	Сост.	Вершины
s1	{0}	s5	{107}	s9	{116}
s2	{5, 23, 58, 75}	s6	{109}	s10	{118}
s3	{36, 88}	s7	{40, 92}	s11	{44, 96}
s4	{ 9, 27, 62, 79}	s8	{125}	s12	{129}

В результате минимизации процессора A68NUM получаются следующие лексические классы:

Таблица 5.2.

Лек. класс	Символы	Лек. класс	Символы	Лек. класс	Символы	Лек. класс	Символы
L1:	d ={ 0–9 }	L2:	{ . }	L3:	{ \ e) }	L4:	{+ – }

и классы эквивалентных состояний (см. табл. 5.3):

[103] Или эквивалентное правило A_{13}: A_{12}^{+}.

Таблица 5.3.

Класс сост.	Состояния класса эквивалентности	Класс сост.	Состояния класса эквивалентности	Класс сост.	Состояния класса эквивалентности
C1	{s1}	C3	{s3}	C6	{s8, s12}
C2	{s2, s4}	C4	{s5, s6}	C7	{s9, s10}
C3	{s3}	C5	{s7, s11 }		

На рис. 5.2 показан вид граф-схемы A68NUM после обработки классов эквивалентных состояний C1–C7 с учетом лексических классов. Заметим, что в этом примере задействованы лишь ситуации (а) и (б) алгоритма минимизации граф-схемы.

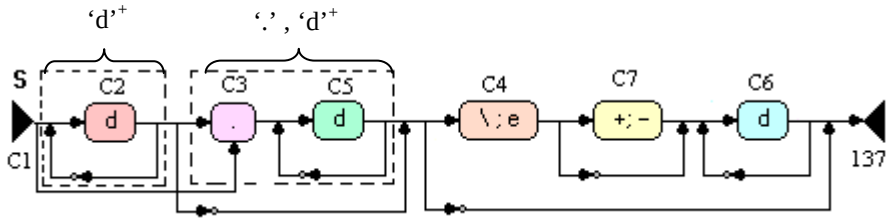


Рис. 5.2. Вид минимизированной граф-схемы A68NUM.

Обратное отображение граф-схемы в RBNF-грамматику

Конвертация граф-схемы в грамматическую форму может быть произведена по-компонентно, т. е. одна компонента граф-схемы отображается в одно правило грамматики. Это делается рекурсивно по виду элементов и структуре их соединений элементов согласно таблицам 1.5 и 1.6 из гл. 1.

***Замечание* 5.** Несмотря на то, что любая RBNF-грамматика отображается в граф-схему, не всякая граф-схема, в частности, полученная в результате минимизации, может оказаться отображаемой в форму RBNF-грамматики *адекватно*, т. е. не существует такой RBNF-грамматики, которая отображалась бы в эту граф-схему. Так в рассматриваемом примере в результате минимизации получилась "слишком минимальная" граф-схема: невозможно построить эквивалентное регулярное выражение с таким же числом операндов, каково число внутренних вершин в граф-схеме. Причина в том, что две подсхемы, представляющие подформулы ('d'+) и ('.', 'd'+), соединены одновременно последовательно (рис. 5.3 а) и параллельно (рис. 5.3 б).

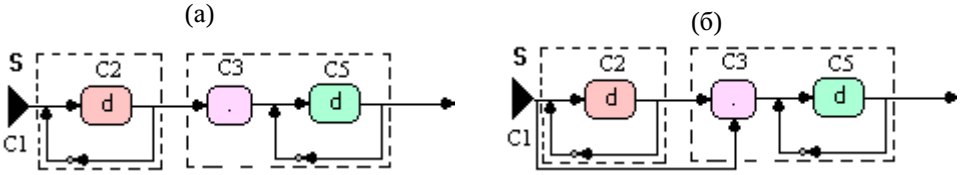


Рис. 5.3. Последовательное и параллельное соединение двух подсхем.

Однако не возможно эти два экземпляра подформул соединить одновременно регулярными операциями конкатенации "," и объединения ";" как операнды одной большой формулы. Действительно, если схема S состоит из подсхем S_1 и S_2, порождающих соответственно множества цепочек L_1 и L_2, а

f_1 и f_2 — формулы, представляющие порождения этих подсхем, то множество L цепочек, порождаемых схемой S, есть

$L = (L_1L_2) \cup (L_1 \cup L_2) = L_1L_2 \cup L_1 \cup L_2 = L_1[L_2] \cup L_2 = [L_1]L_2 \cup L_1$.

Соответствующее регулярное выражение f, представляющее множество L, есть $f = f_1 , [f_2] ; f_2$ или $f = [f_1] , f_2 ; f_1$.

Если схема S не порождает множество цепочек, в котором содержится пустая цепочка, то регулярное выражение, представляющее такую схему, не адекватно ей. В противном случае соответствующее регулярное выражение есть $f = [f_1], [f_2]$, и оно адекватно схеме S, т. е. отображается в ту самую схему, из которой оно было получено.

Именно такой случай имеет место в языке Паскаль. В нем числа описываются подобной редуцированной граф-схемой (см. рис. 5.4), но без параллельной связи, о которой шла речь выше (см. подсхемы **F** и **E**). Поэтому эта граф-схема адекватна регулярному выражению ‘d’$^+$, [‘.’, ‘d’$^+$], [‘e’, [‘+’ ; ‘–’], ‘d’$^+$], которое получается из нее обратным отображением.

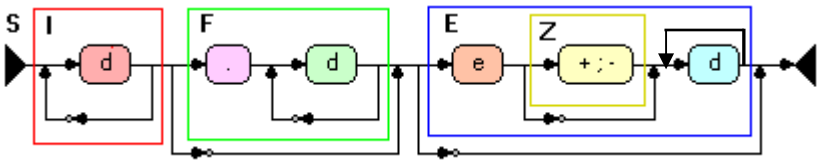


Рис. 5.4. Редуцированная граф-схема чисел языка Паскаль.

Пример 5.2. Редукция грамматики выражений

Пусть дана исходная КС-грамматика EXPRESS**:**

NONTERMINALS: S, E, T, F.
TERMINALS : ‘+’, ‘*’, ‘(‘, ‘)’, ‘*a*’.
(1) S : E.
(2) E : E, '+', T ; T.
(3) T : T, '*', F; F.
(4) F : '(', E, ')' ; '*a*'.

В ней все нетерминалы, кроме начального S, являются самовставленными, и потому не исключаются методом, описанном в алгоритме 5.1. Однако исключение из данной грамматики нетерминалов T и F возможно, несмотря на то, что они самовставленные.

Действительно, после исключения явных левосторонних рекурсий (посредством функции Reduce ТК SYNTAX) из правил (2) и (3) мы получаем регуляризованную RBNF-грамматику EXPRESS1:

NONTERMINALS: S, E, T, F.

(1) S: E.
(2) E: T , ('+' , T)*.
(3) T: F , ('*' , F)*.
(4) F: '(' , E , ')' ; '*a*'.

Затем без труда обнаруживается, что S = ρ_1(E), E = ρ_2(T), T = ρ_3(F), F = ρ_4(E), где ρ_i (i = 1, 2, 3, 4) — некоторые регулярные выражения, зависящие от нетерминалов, указанных в скобках. Это значит, что E = ρ_2(T) = $\rho_2(\rho_3$(F)) = $\rho_2(\rho_3(\rho_4$(E))) и для автоматического выполнения указанных подстановок достаточно символы T и F перевести в список вспомогательных понятий, не меняя множества правил. В результате получаем регуляризованную RBNF-грамматику EXPRESS2:

NONTERMINALS: S, E.

AUXILIARY NOTIONS : T, F.

(1) S: E. (3) T: F , ('*' , F)*.

(2) E: T , ('+' , T)*. (4) F: '(' , E , ')' ; '*a*'.

При построении граф-схемы по этой грамматике регулярные выражения, определяющие вспомогательные понятия T и F, встраиваются в компоненту для нетерминала E (см. рис. 5.5).

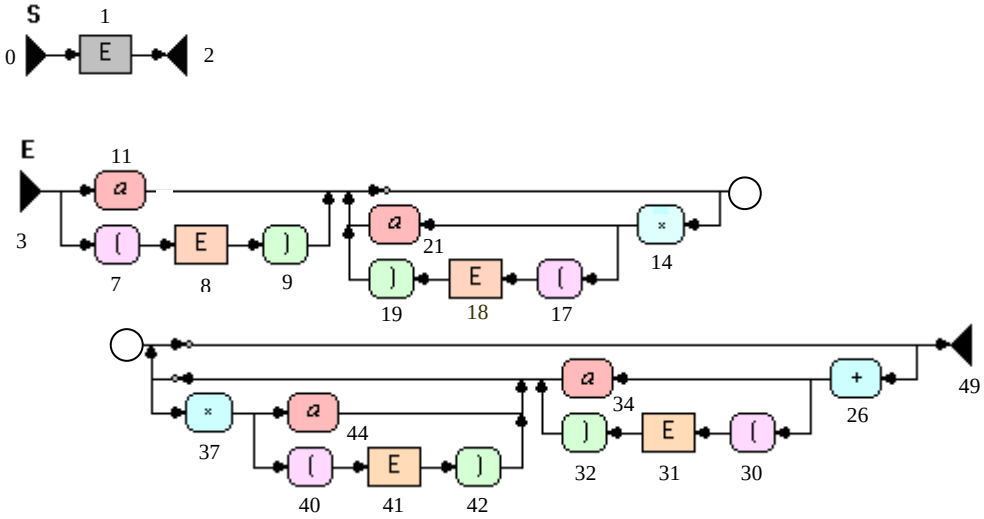


Рис.5.5. Граф-схема грамматики EXPRESS2.

Процессор, построенный по исходной граф-схеме EXPRESS2, имеет характеристики состояний, указанные в табл. 5.4.

Таблица 5.4.

Сост.	**Вершины**	**Сост.**	**Вершины**	**Сост.**	**Вершины**	**Сост.**	**Вершины**
s1	{0}	s6	{30}	s11	{40}	**s16**	{18}
s2	{7}	s7*	{34}	s12*	{44}	**s17**	{41}
s3*	{11}	s8	{17}	**s13**	{1}	s18*	{9}
s4	{26}	s9*	{21}	**s14**	{8}	s19*	{32}
s5	{14}	s10	{37}	**s15**	{31}	s20*	{19}
Звездочками помечены подавляемые состояния. Жирным шрифтом выделены возвратные состояния.						s21*	{42}

Оптимизация процессора EXPRESS2 дает классы эквивалентных входных символов и состояний, приведенных в табл. 5.5 и 5.6.

Таблица 5.5.

Лек. класс	Симв.	Лек. класс	Симв.	Лек. класс	Симв.	Лек. класс	Симв.
L1:	{ + * }	L2:	{ (}	L3:	{) }	L4:	{ a }

Таблица 5.6.

Класс	Состояния класса	Множества вершин
C1	{s1}	{{0}}
C2	{s2, s6, s8, s11}	{{7-(},{30-(},{17-(},{40-(}}
C3*	{s3, s7, s9, s12, s18, s19, s20, s21}	{{11-a},{34-a},{21-a},{44-a}, {9-)},{32-)},{19-)},{42-)}}
C4	{s4, s5, s10}	{{26-+},{14 -*},{37-*}}
C5	{s13}	{{1-E}}
C6	{s14, s15, s16, s17 }	{{8-E},{31-E},{18-E},{41-E}}

Применение алгоритма 5.2 с учетом информации, приведенной в табл. 5.4–5.6, дает минимизированную граф-схему, показанную на рис. 5.6.

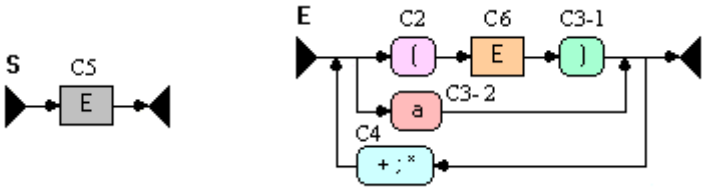


Рис. 5.6. Минимизированную граф-схема EXPRESS2.

Поясним, что класс подавляемых состояний C3 породил два образа вершин, отмеченных на рис. 5.6 обозначениями C3-1 и C3-2.

Реконструкция RBNF-грамматики по полученной редуцированной граф-схеме EXPRESS2 дает только два правила для нетерминалов S и E:

NONTERMINALS: S, E.

(1) S : E.

(2) E : (‘(’, E, ‘)’; ‘*a*’) # (‘+ ; *’).

Это и есть адекватная минимальная RBNF-грамматика, эквивалентная исходной КС-грамматике EXPRESS. В ней роль терминалов играют синтаксические классы входных символов. Правило (1) не изменилось. Оно используется лишь для того, чтобы удовлетворить одному из условий SYNTAX-технологии: начальный нетерминал грамматики не должен встречаться в правой части ни одного из ее правил.

Глава 6

ПРАКТИЧЕСКИЕ АСПЕКТЫ ПРИМЕНЕНИЯ SYNTAX-ТЕХНОЛОГИИ

6.1. ПОСТРОЕНИЕ ТРАНСЛЯЦИОННОЙ ГРАММАТИКИ

TSL[104]-спецификацию трансляции некоторого языка программирования можно построить, исходя из обычной BNF-грамматики, которая является традиционной формой представления синтаксиса языка при его публикации.

Как известно, каждое правило BNF-грамматики состоит из левой и правой частей, разделяемых метасинтаксической связкой '::='. В левой части находится определяемый нетерминал, а правая есть список альтернатив, разделяемых метасинтаксической связкой '|'. Альтернатива — это цепочка членов, каждый из которых есть терминал или нетерминал. Нетерминалы принято представлять цепочками символов, заключенными в метасинтаксические скобки '<' и '>', а все прочие символы, отличающиеся от метасинтаксических знаков, трактуются как терминальные. Например, следующие правила BNF-грамматики определяют целые без знака — традиционные элементы многих языков программирования:

<целое без знака> ::= <цифра> | <целое без знака> <цифра>

<цифра> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

В этих правилах <целое без знака> и <цифра> — нетерминалы, а $0, \ldots, 9$ — терминалы. Имеются две альтернативы в первом правиле и десять — во втором. Вторая альтернатива первого правила состоит из двух (конкатенируемых) членов <целое без знака> и <цифра>, каждый из которых является нетерминалом. В TSL-нотации эти же правила имеют вид

целое без знака : цифра ; целое без знака , цифра.

цифра : '0' ; '1' ; '2' ; '3' ; '4' ; '5' ; '6' ; '7' ; '8' ; '9'.

Использование в ней явного обозначения для конкатенации (запятой) позволяет опускать метасинтаксические скобки в обозначении нетерминалов. То, что эти цепочки обозначают нетерминалы, определяется благодаря дополнительному описанию:

Nonterminals : целое без знака, цифра.

включаемому в TSL-спецификацию.

[104] Метаязык TSL описывается в гл. 8.

Для определения операционной семантики некоторой конструкции языка достаточно в правые части правил исходной грамматики внести семантические и резольверные символы и задать их интерпретацию в некоторой операционной среде на некотором инструментальном языке программирования. Планирование семантических преобразований производится в контексте отдельного правила грамматики, точнее, каждой его альтернативы (т.е. члена объединения). Следует исходить из того, что входная цепочка реализуемого языка является заданием преобразований операционной среды, которые выполняются по ходу ее сканирования и анализа.

Предполагается, что механизм анализа, сканируя входную цепочку, сопоставляет некоторую ее часть с рассматриваемой альтернативой, как с образцом, определяющим ее структуру. При этом подразумевается, что те преобразования, которые заданы подцепочками, соответствующими вхождениям нетерминалов в образец, уже выполнены во время сканирования этих подцепочек. В данной же альтернативе правила необходимо определить продолжение этих преобразований операционной среды, исходя из текущего ее состояния.

Представление о текущем состоянии операционной среды складывается при рассмотрении данной альтернативы по ее составу и правилам трансляционной грамматики, определяющим нетерминалы, входящие в данную альтернативу, с учетом семантик, используемых в упомянутых правилах.

Продолжение преобразований, ассоциированных с некоторым вхождением нетерминала в данную альтернативу, задается путем вставки семантического символа или даже последовательности таких символов *после* данного вхождения нетерминала (т.е. справа) — для прямого просмотра или *перед* ним (т.е. слева) — для обратного.

В отличие от описанной трактовки нетерминалов, предполагается, что с вхождением терминала в образец не связаны никакие предварительные преобразования. Они как раз и должны начаться в данной альтернативе в связи с данным вхождением терминального символа. Для указания требуемых преобразований, связанных с данным вхождением терминала, следует вставить семантический символ или последовательность таких символов *слева* от него — для прямого просмотра, или *справа* от него — для обратного.

Что касается резольверных символов, то размещение резольверной цепочки слева от терминального символа (для прямого просмотра) или справа от него (для обратного) определяет некоторое контекстное условие[105], при выполнении которого только и допустим данный символ.

Каждый новый резольверный или семантический символ полезно вносить в реестр спецификаций, ассоциированных с такими символами предикатов и преобразований операционной среды. Ведение такого реестра в процессе написания трансляционной грамматики помогает повторно использовать уже определенные семантики и резольверы в других правилах. В дальнейшем эти спецификации используются при написании кода, реализующего соответствующие предикаты и преобразования операционной среды.

[105] Это условие — истинность конъюнкции предикатов, ассоциированных с резольверными символами соответственно прямого или обратного просмотров.

Конструкция <целое без знака>. Для примера определим операционную семантику конструкции <целое без знака>, описанной двумя вышеприведенными правилами, включив в правила грамматики несколько семантических символов следующим образом[106]:

```
{ Список нетерминалов}
Nonterminals : целое без знака , цифра .

{ Список семантик прямого просмотра}
Forward pass semantics : Init, Append, D0, D1, D2, D3, D4, D5, D6, D7, D8, D9.

{ Правила}
целое без знака : цифра , Init ; целое без знака , цифра , Append.
цифра : D0 , '0' ; D1 , '1' ; D2 , '2' ; D3 , '3' ; D4 , '4' ;
        D5 , '5' ; D6 , '6' ; D7 , '7' ; D8 , '8' ; D9 , '9' .

{ Описание операционной среды (на языке Паскаль) }
ENVIRONMENT

var UnsignedInteger, Digit : Integer;

IMPLEMENTATION
        { Интерпретация сематических символов}
procedure Init;
begin UnsignedInteger := Digit end;
procedure Append;
begin UnsignedInteger := UnsignedInteger*10+Digit end;
procedure D0; begin Digit := 0 end;
procedure D1; begin Digit := 1 end;
procedure D2; begin Digit := 2 end;
procedure D3; begin Digit := 3 end;
procedure D4; begin Digit := 4 end;
procedure D5; begin Digit := 5 end;
procedure D6; begin Digit := 6 end;
procedure D7; begin Digit := 7 end;
procedure D8; begin Digit := 8 end;
procedure D9; begin Digit := 9 end;
```

Очевидно, что эта трансляционная грамматика определяет "сборку" значения целочисленной переменной UnsignedInteger по данной цепочке цифр на входе.

Принцип планирования семантической обработки входа достаточно естественен. Семантики D0–D9, используемые во втором правиле, реализуют преобразование текущей входной литеры цифры в эквивалентное арифметическое значение переменной Digit. Первая альтернатива первого правила определяет

[106] Этот фрагмент спецификации записан на языке TSL.

сборку одноразрядного целого из уже определенного значения переменной Digit, а вторая альтернатива определяет сборку многоразрядного целого из уже частично собранного значения переменной UnsignedInteger и еще одной цифры, арифметическое значение которой уже зафиксировано переменной Digit.

Таким образом, этот пример иллюстрирует основной принцип планирования семантических преобразований в тексте грамматики: любая конструкция языка (например, целое без знака) продолжает действия своих подконструкций (цифра и целое без знака), используя их результаты, зафиксированные в операционной среде (а именно, переменными Digit и UnsignedInteger).

Заметим, что в компилирующих системах входная цепочка (программа) неявно[107] задает процесс ее перевода в соответствующую выходную цепочку (программу), которая интерпретируется (выполняется) аппаратурой. Соответственно спецификация компилятора в виде трансляционной грамматики использует семантики, которые задают процесс компиляции. С другой стороны, в интерпретирующих системах входная цепочка неявно задает процесс ее исполнения как программы, т.е. процесс обработки входных данных. Соответственно, в этом случае семантики данных реализуют действия над данными.

Следует заметить, что построенная нами спецификация не годится для непосредственного использования в SYNTAX-технологии, так как лежащая в ее основе управляющая грамматика леворекурсивна. Наилучший для технологии способ исключить левую рекурсию — заменить ее итерацией. Тогда первое правило примет вид:

целое без знака : цифра , Init , (цифра , Append)*.

В таком виде управляющая грамматика (при том же описании операционной среды) может быть использована для построения сплайнового процессора, реализующего сборку целых чисел. Если, кроме того, перевести символ ‘цифра’ из алфавита нетерминалов в алфавит вспомогательных понятий, используя описание

Auxiliary notions : цифра .

то результирующий процессор будет конечным.

Совершенствование этой спецификации можно продолжить путем дальнейшего эквивалентного преобразования правил и описания операционной среды. Действительно, если унифицировать обработку всех цифр посредством одной новой семантики D, реализуемой процедурой:

```
procedure D;
var s : string;
begin s := LR; Digit := ord (s[1]) – ord (‘0’) end;
```

и заодно не делать исключения для первой цифры целого без знака, а обрабатывать ее как все последующие цифры при помощи семантики Append, для чего потребуется новая процедура инициализации процесса сборки:

```
procedure SetZero; begin UnsignedInteger := 0 end;
```

то окончательная версия спецификации имеет следующий вид:

[107] Эти действия становятся явными в управляющей цепочке, регенерируемой по данной входной цепочке в процессе ее синтаксического анализа.

Nonterminals : целое без знака.
Auxiliary notions : цифра.
Forward pass semantics : SetZero , Append, D .
целое без знака : SetZero, (цифра , Append)$^{+}$.
цифра : D , ('0' ; '1' ; '2' ; '3' ; '4' ; '5' ; '6' ; '7' ; '8' ; '9') .

ENVIRONMENT

var UnsignedInteger, Digit : Integer;

IMPLEMENTATION

procedure SetZero; **begin** UnsignedInteger := 0 **end**;
procedure Append;
begin UnsignedInteger := UnsignedInteger * 10 + Digit **end**;
procedure D;
var s : string;
begin s := LR; Digit :=ord (s[1]) – ord ('0') **end**;

В процедуре D используется встроенная функция LR, поставляющая строковое представление входной лексемы.

Синтаксически управляемый калькулятор. В качестве более сложного примера приведем первоначальную версию трансляционной грамматики, задающей интерпретацию арифметических выражений над числами[108]. Предполагается, что числа — целые и вещественные, представляются в формате языка Паскаль, а в качестве операций допустимы унарные и бинарные '+' и '–', а также умножение '*' и деление '/'. Как обычно, круглые скобки используются для явного указания порядка вычислений.

Система семантик этой грамматики обеспечивает реализацию алгоритма Дейкстры преобразования выражений в обратную польскую запись, который совмещен с алгоритмом вычисления выражения, уже представленного в такой форме. Напомним, что выражения в обратной польской записи вычисляются в текстуальной последовательности. Поскольку выполнению операции должно предшествовать вычисление операндов, то в обратной польской форме операция следует после своих операндов[109]. Таким образом, выражение вида *A+B* в обратной польской форме имеет вид *AB+*. Разумеется, скобки, которые используются в выражении для явного указания порядка вычислений, в обратной польской форме исчезают, поскольку в ней последовательность вычислений, как уже отмечалось, определяется текстуальным порядком. Операнды *A* и *B* могут быть числами или выражениями. В последнем случае подразумевается, что они также представлены в обратной польской форме.

---

[108] Окончательная версия этой грамматики, приведенная в разд. 1.3, была получена путем эквивалентных преобразований этой первоначальной версии. Они описываются в следующем разделе. Эквивалентным преобразованиям трансляционных грамматик также посвящена гл. 5.

[109] Поэтому ее часто называют *постфиксной записью* в отличие от традиционной — *инфиксной*.

Алгоритм Дейкстры состоит в следующем. Входное выражение просматривается слева направо, причем операнды немедленно выдаются на выход в обратную польскую форму, а операции попадают в нее лишь после их упорядочивания в соответствии с их старшинством и с учетом расстановки скобок. Для упорядочивания операций используется магазин операций. Именно, любая операция на входе алгоритма сравнивается по старшинству с операцией на вершине магазина, и если последняя имеет старшинство, не меньшее, чем операция на входе, то операция с вершины магазина передается на выход алгоритма, т.е. в обратную польскую запись. Такая разгрузка магазина операций происходит до тех пор, пока на его вершине не обнаружится операция меньшего старшинства, чем операция на входе, или пока на вершине не появится открывающая скобка или магазин не окажется пуст. Только после этого входная операция записывается в магазин.

Скобки на входе алгоритма обрабатываются так: открывающая скобка немедленно записывается в магазин, а закрывающая вызывает разгрузку магазина операций до тех пор, пока на его вершине не покажется открывающая скобка. Она просто сбрасывается с вершины, после чего продолжается просмотр входного выражения.

Когда входное выражение просматривается до конца, алгоритм завершается тем, что все оставшиеся в магазине операции извлекаются из него и исполняются над соответствующими операндами в магазине операндов.

При вычислении выражения, представленного в обратной польской форме, используется магазин операндов. Именно, когда при чтении обратной польской формы слева направо встречается операнд, его значение записывается в магазин операндов. Когда на входе встречается операция, она выполняется над верхними значениями в магазине операндов, и результат замещает значения операндов. При завершении этого процесса результат образуется на вершине магазина. Он является единственным его элементом.

Для совмещения вычислений с преобразованием в обратную польскую форму алгоритм Дейкстры модифицирован: операнды не посылаются на выход в обратную польскую запись, а помещаются в магазин операндов. Аналогично, и операции не посылаются в обратную польскую запись, а выполняются над верхними значениями в магазине операндов.

В приведенной далее грамматике, записанной на метаязыке TSL, нетерминалы представлены акронимами — английскими терминами, используемыми в правилах грамматики, и синонимами — терминами на русском языке, которые используются в диагностических сообщениях, автоматически генерируемых системой SYNTAX. Вслед за правилами управляющей грамматики описывается операционная среда, в которой интерпретируются семантики.

CALC — трансляционная грамматика калькулятора (первоначальный вариант).

Nonterminals:

PROGRAM (ПРОГРАММА),
EXPRESSION (ВЫРАЖЕНИЕ),
TERM (СЛАГАЕМОЕ),
FACTOR (МНОЖИТЕЛЬ),
NUMBER (ЧИСЛО),
INTEGER NUMBER (ЦЕЛОЕ_ЧИСЛО),
REAL NUMBER (ВЕЩЕСТВЕННОЕ_ЧИСЛО),
FRACTION (ДРОБЬ),
FRACTIONAL PART (ДРОБНАЯ_ЧАСТЬ),
EXPONENT PART (ПОРЯДОК),
EXPONENT (ВЕЛИЧИНА_ПОРЯДКА),
DIGIT (ЦИФРА),
SIGN (ЗНАК).

Forward pass semantics :

Reset, {Инициирует вычисление выражения}
Complete, {Завершает разгрузку магазина операций и вычисление выражения}
Push Monadic Plus, {Помещает в магазин операций унарный '+'}
Push Monadic Minus, {Помещает в магазин операций унарный '–'}
Push Priority 1 Plus, { Обработка бинарной операции '+' приоритета 1}
Push Priority 1 Minus, {Обработка бинарной операции '–' приоритета 1}
Push Priority 2 Mult, {Обработка операции '*' приоритета 2}
Push Priority 2 Div , {Обработка операции '/' приоритета 2}
Push Operand, {Помещает значение операнда в магазин операндов}
Push Open Par, {Помещает '(' в магазин операций}
UnloadAndDiscardOpenPar, {Разгрузка операций с вершины магазина операций до '('}
Set Integer Number, {Сбока цисла из целого}
Set Real Number, {Сбока цисла из вещественного}
Init Integer Number, {Инициализация целого по первой цифре}
Append Digit, {Включение очередной цифры в целое}
Set Digit 0, {Приобразование литеры 0 в целое значение}
Set Digit 1, {Приобразование литеры 1 в целое значение}
Set Digit 2, {Приобразование литеры 2 в целое значение}
Set Digit 3, {Приобразование литеры 3 в целое значение}
Set Digit 4, {Приобразование литеры 4 в целое значение}
Set Digit 5, {Приобразование литеры 5 в целое значение}
Set Digit 6, {Приобразование литеры 6 в целое значение}
Set Digit 7, {Приобразование литеры 7 в целое значение}
Set Digit 8, {Приобразование литеры 8 в целое значение}
Set Digit 9, {Приобразование литеры 9 в целое значение}

```
Init Real with Fraction,     {Инициализация вещественного числа десятичной дробью}
Init Real with Integer,      {Инициализация вещественного числа целым}
Append Exponent Part,        {Учет порядка}
Init Fraction,               {Инициализация десятичной дроби целой частью}
Append Fractional Part, {Добавление дробной части к десятичной дроби}
Set Fractional Part,         {Установка дробной части}
Set Exponent Part,           {Установка порядка}
Set Unsigned Exponent,       {Установка величины порядка}
        Set Signed Exponent,     {Установка величины порядка со знаком}
        Set Positive,            {Установка знака порядка '+' }
Set Negative. {Установка знака порядка '–'}
        {Правила}
PROGRAM : Reset , EXPRESSION , Complete ,  'EOF'.                          (1)
EXPRESSION: TERM ;                                                         (2.1)
    EXPRESSION, Push Priority 1 Plus, '+', TERM;                           (2.2)
    EXPRESSION, Push Priority 1 Minus, '–', TERM.                          (2.3)
TERM : FACTOR ;                                                            (3.1)
    TERM , Push Priority 2 Mult , '*' , FACTOR ;                           (3.2)
    TERM , Push Priority 2 Div ,  '/' , FACTOR.                            (3.3)
FACTOR : NUMBER , Push Operand;                                            (4.1)
    Push Open Par, '(', EXPRESSION , Unload And Discard Open Par, ')';     (4.2)
    Push Monadic Plus , '+' , FACTOR ;                                     (4.3)
    Push Monadic Minus , '–' , FACTOR.                                     (4.4)
NUMBER : INTEGER NUMBER , Set Integer Number ;                             (5.1)
    REAL  NUMBER , Set Real Number.                                        (5.2)
INTEGER NUMBER : DIGIT, Init Integer Number;                               (6.1)
    INTEGER NUMBER , DIGIT, Append Digit.                                  (6.2)
DIGIT : Set Digit 0, '0'; Set Digit 1, '1'; Set Digit 2, '2';
    Set Digit 3, '3'; Set Digit 4, '4'; Set Digit 5, '5';
    Set Digit 6, '6'; Set Digit 7, '7'; Set Digit 8, '8';
    Set Digit 9, '9'.                                                      (7)
REAL NUMBER : FRACTION , Init Real with Fraction;                          (8.1)
    FRACTION, Init Real with Fraction,
        EXPONENT PART, Append Exponent Part;                               (8.2)
    INTEGER NUMBER, Init Real with Integer,
        EXPONENT PART, Append Exponent Part.                               (8.3)
FRACTION : INTEGER NUMBER, InitFraction, FRACTIONAL PART,
    AppendFractionalPart .                                                 (9)
FRACTIONAL PART : '.' , INTEGER NUMBER , Set Fractional Part .             (10)
EXPONENT PART : 'E' ,  EXPONENT, Set Exponent Part .                       (11)
EXPONENT : INTEGER NUMBER , Set UnsignedExponent ;                         (12.1)
    SIGN, INTEGER NUMBER, Set Signed Exponent.                             (12.2)
SIGN : Set Positive, '+';                                                  (13.1)
      Set Negative, '–'.                                                   (13.2)
```

```
ENVIRONMENT
const Size = 100;  {Размер магазинов операндов и операций }
type TOperation = ( MonadicPlus, MonadicMinus, Plus,
                    Minus, Mult, Division, OpenPar);
TPrio = 0 .. 3;
TOperationStackItem =    record
                            Operation : TOperation;
                            Priority : TPrio
                         end;
TOperandStack = array [1..Size] of real; {Магазин операндов}
TOperationStack = array [1..Size] of TOperationStackItem; {Магазин операций}
var  OperandStack : TOperandStack;
     OperationStack : TOperationStack;
     OperandStackTop, OperationStackTop : 0 .. Size;
     Number,  RealNumber ,
     Fraction, FractionalPart, ExponentPart: real;
     IntegerNumber, DigitsNmb, Exponent,  Sign : integer;
          { Впомогательные процедуры и функции}
function Power ( A , B : integer ) : real; {Вычисляет A^B}
var C : real; I : integer;
begin C := 1; for I := 1 to B do C := C * A; Power := C end;
procedure Unload ( Prio : TPrio ); {Разгрузка и выполнение операций
                                    из магазина операций}
 function Unloading : Boolean; {Дает true, если разгрузка из
                                магазина операций возможна}
 begin  {Unloading}
        Unloading := ( OperationStackTop > 0 ) and
        (OperationStack[OperationStackTop].Operation <> OpenPar) and
        (Prio <= OperationStack[ OperationStackTop ].Priority);
 end {Unloading};
begin {Unload}
 while Unloading do
with OperationStack[OperationStackTop] do
begin
case Operation  of
      MonadicPlus: {Skip};
      MonadicMinus: OperandStack[OperandStackTop] :=
                          -OperandStack[OperandStackTop]
else
begin
   case Operation  of
            Mult:     OperandStack [OperandStackTop-1] :=
                          OperandStack [OperandStackTop-1]*
                          OperandStack [OperandStackTop];
                      Division:     OperandStack [OperandStackTop-1] :=
                          OperandStack [OperandStackTop-1]/
                          OperandStack [OperandStackTop];
                      Plus:         OperandStack[OperandStackTop-1] :=
                          OperandStack [OperandStackTop-1]+
                          OperandStack [OperandStackTop];
```

```
                    Minus:    OperandStack[OperandStackTop-1]:=
                              OperandStack[OperandStackTop-1]-
                              OperandStack[OperandStackTop]
   end {case};
Dec (OperandStackTop)
end
end {case};
Dec (OperationStackTop)
end
end {Unload};
          { Реализация семантик}
procedure Reset; {Инициализирует вычисление входного выражения}
begin  {Опустошение магазинов}
  OperationStackTop := 0; OperandStackTop := 0;
end;
procedure Complete; {Завершает разгрузку магазина операций
                     и вычисление выражения}
begin
 Unload(0); {Разгружает магазин операций, завершая вычисление выражения}
 PrintReal (OperandStack [OperandStackTop]); {Печать результата}
 Dec (OperandStackTop) {Сбос результата выражения
                        с вершины магазина операндов}
end;
procedure PushPriority1Plus; {Обработка бинарного '+'}
begin
 Unload(1); {Разгружает из магазина и выполняет операции приоритета 1 и выше}
 Inc(OperationStackTop); {Помещает в магазин текущую операцию на входе}
 with OperationStack[OperationStackTop] do
 begin Operation := Plus; Priority := 1 end
end;
procedure PushPriority1Minus; {Обработка бинарного '–'}
begin
 Unload(1); {Разгружает из магазина и выполняет операции приоритета 1 и выше}
 Inc(OperationStackTop); {Помещает в магазин текущую операцию на входе}
 with OperationStack[OperationStackTop] do
 begin Operation := Minus; Priority :=1 end
end;
procedure PushPriority2Mult; {Обработка '*'}
begin
Unload(2); {Разгружает из магазина и выполняет операции приоритета 2 и выше}
Inc ( OperationStackTop ); {Помещает в магазин текущую операцию на входе}
with OperationStack[OperationStackTop] do
begin Operation := Mult; Priority := 2 end
end;
procedure PushPriority2Div; {Обработка '/'}
begin
 Unload(2);{Разгружает из магазина и выполняет операции приоритета 2 и выше}
 Inc ( OperationStackTop ); {Помещает в магазин текущую операцию на входе}
```

```
with OperationStack[OperationStackTop] do
begin Operation := Division; Priority := 2 end
end;
procedure PushMonadicPlus; {Помещает унарный плюс в магазин операций}
begin Inc ( OperationStackTop );
with OperationStack[OperationStackTop] do
begin Operation:=MonadicPlus; Priority:=3 end
end;
procedure PushMonadicMinus; {Помещает унарный минус в магазин операций}
begin Inc ( OperationStackTop );
with OperationStack[OperationStackTop] do
begin Operation := MonadicMinus; Priority := 3 end
end;
procedure PushOperand; {Помещает значение операнда в магазин операндов}
begin Inc(OperandStackTop);
OperandStack[OperandStackTop]:= Number
end;
procedure PushOpenPar; {Помещает '(' в магазин операций}
begin Inc( OperationStackTop );
with OperationStack[OperationStackTop] do
 begin Operation := OpenPar; Priority:=0 end
end;
procedure UnloadAndDiscardOpenPar; {Разгузка магазина операций до
                                    открывающей скобки}
begin Unload(0); Dec(OperationStackTop) end;
procedure SetIntegerNumber; {Сборка числа из целого}
begin  Number := IntegerNumber end;
procedure SetRealNumber; {Сборка числа из вещественного}
begin Number := RealNumber end;
procedure InitIntegerNumber; {Инициализация целого по первой цифре}
begin IntegerNumber := Digit ; DigitsNmb := 1 end;
procedure Append Digit; {Включение очередной цифры в целое}
begin IntegerNumber := IntegerNumber*10+Digit ; Inc ( DigitsNmb ) end;
          { Трансформация литер цифр в целочисленный эквивалент}
procedure SetDigit0; begin Digit := 0 end;
procedure SetDigit1; begin Digit := 1 end;
procedure SetDigit2; begin Digit := 2 end;
procedure SetDigit3; begin Digit := 3 end;
procedure SetDigit4; begin Digit := 4 end;
procedure SetDigit5; begin Digit := 5 end;
procedure SetDigit6; begin Digit := 6 end;
procedure SetDigit7; begin Digit := 7 end;
procedure SetDigit8; begin Digit := 8 end;
procedure SetDigit9; begin Digit := 9 end;
procedure InitRealWithFraction; {Инициализация вещественного числа
                                 десятичной дробью}
begin RealNumber := Fraction end;
```

```
procedure InitRealWithInteger; {Инициализация вещественного числа целым}
begin RealNumber := IntegereNumber end;
procedure AppendExponentPart; {Учет порядка}
begin RealNumber := RealNumber * ExponentPart end;
procedure InitFraction; {Инициализация десятичной дроби целой частью}
begin Fraction := IntegerNumber end;
procedure AppendFractionalPart; {Добавление дробной части к десятичной дроби}
begin Fraction := Fraction + FractionalPart end;
procedure SetFractionalPart; {Установка дробной части}
var  i : integer;
begin FractionalPart := IntegerNumber;
for i := 1 to DigitsNmb do FractionalPart := FractionalPart * 0.1
end;
procedure SetExponentPart; {Установка  порядка}
begin
if Exponent >= 0
then  ExponentPart := Power(10 , Exponent)
else  ExponentPart := 1 / Power (10 , – Exponent)
end;
procedure SetUnsignedExponent; { Установка абсолютной величины порядка}
begin Exponent  := IntegerNumber end;
procedure SetSignedExponent; {Установка величины порядка с учетом знака}
begin Exponent := IntegerNumber * Sign end;
procedure SetPositive; {Установка знака порядка '+'}
begin Sign := 1 end;
procedure SetNegative; { Установка знака порядка '–'}
begin Sign := –1 end;
```

Данная спецификация синтаксически управляемого калькулятора может рассматриваться лишь как первое приближение к рабочей версии, пригодной для использования в SYNTAX-технологии. Последняя может быть получена за счет эквивалентных преобразований этой спецификации, описываемых в следующем разделе.

6.2. ЭКВИВАЛЕНТЫЕ ПРЕОБРАЗОВАНИЯ ТРАНСЛЯЦИОННОЙ ГРАММАТИКИ

Как отмечалось в гл. 5, SYNTAX-технология требует использования трансляционных грамматик, которые обладают некоторыми "хорошими" свойствами. В той же главе были описаны некоторые способы эквивалентных преобразований трансляционных грамматик, которые могут использоваться, если данная грамматика окажется не вполне подходящей. Например, в грамматике CALC, описанной в предыдущем разделе, встречаются случаи левой рекурсии, которая не допускается SYNTAX-технологией. Покажем, какие эквивалентные преобразования трансляционной грамматики CALC достаточно выполнить для того, чтобы сделать ее пригодной для построения процессора, реализующего трансляцию, которую она определяет.

Грамматика CALC. Последовательные эквивалентные версии регулярных выражений в правых частях правил будем разделять знаком '='. Начнем преобразования с правила (2), поскольку в нем имеются две леворекурсивные альтернативы: (2.2) и (2.3). Итак, имеем:

```
EXPRESSION = TERM ;
                EXPRESSION , Push Priority 1 Plus , '+' ,TERM;
                EXPRESSION , Push Priority 1 Minus , '–' ,TERM
             = TERM, ((Push Priority 1 Plus , '+' ;
               Push Priority 1 Minus , '–' ) , TERM)*
             = TERM # (Push Priority 1 Plus , '+' ;
               Push Priority 1 Minus , '–' )
             = TERM # (Push Priority 1 Operation , ( '+' ; '–' )).
```

Здесь на последнем шаге преобразований вместо двух различных семантик Push Priority 1 Plus и Push Priority 1 Minus, каждая из которых специализирована на обработку соответствующей бинарной операции '+' или '–', введена новая унифицированная семантика Push Priority 1 Operation, которая обрабатывает любую из этих двух бинарных операций. Однако платой за такую унификацию является явная зависимость реализации этой новой семантики от текущего входного символа ('+' или '–').

Другими словами, если в исходной версии грамматики в зависимости от текущего входного символа было запланировано выполнение соответствующей семантики, предназначенной для обработки именно этого символа, то в новой версии в обоих случаях выполняется одна и та же семантика, которая, однако, вынуждена сама анализировать (повторно) текущий входной символ.

Новая семантика имеет следующую реализацию:

```
procedure PushPriority1Operation; {Обработка бинарного '+' или '–'}
begin Unload(1); {Разгружает из магазина и выполняет операции
                  приоритета 1 и выше}
 Inc(OperationStackTop); {Помещает в магазин текущую операцию на входе}
with OperationStack[OperationStackTop] do
begin  case CurrentSymbol of
        '+' : Operation := Plus;
        '–' : Operation := Minus
              end;
        Priority := 1
end
end;
```

Предполагается, что CurrentSymbol — вспомогательная функция, выдающая текущий входной символ.

Вполне аналогичные преобразования правила (3) дают следующий результат:

```
TERM : FACTOR # (Push Priority 2 Operation , ( '*' ; '/' )) .
```

Новая семантика Push Priority2 Operation имеет следующую реализацию:

```
procedure PushPriority2Operation; {Обработка бинарного '*' или '/'}
begin
 Unload(2); {Разгружает из магазина и выполняет операции приоритета 2 и выше}
 Inc(OperationStackTop); {Помещает в магазин текущую операцию на входе}
 with OperationStack[OperationStackTop] do
  begin
    case CurrentSymbol of
            '*' : Operation := Mult;
            '/' : Operation := Division
    end; Priority := 2
  end
end;
```

Подставляя в новое правило для нетерминала EXPRESSION только что полученное выражение для нетерминала TERM и вводя вспомогательное понятие DYADIC OPERATION при помощи правила

```
DYADIC OPERATION : Push Priority 1 Operation, ( '+' ; '–' ) ;
                   Push Priority 2 Operation, ( '*' ; '/' ) .
```

получаем следующее правило для нетерминала EXPRESSION:

```
EXPRESSION : FACTOR # DYADIC OPERATION.
```

Обработку бинарных операций разного старшинства в правиле для вспомогательного понятия DYADIC OPERATION также можно унифицировать при помощи новой семантики:

```
procedure PushDyadicOperation; {Обработка бинарных операций}
var Prio : TPrio; CS : Char;
begin
CS := CurrentSymbol;
case CS of
        '+', '–' : Prio := 1;
        '*' , '/' : Prio := 2
end;
Unload (Prio); Inc(OperationStackTop);
with OperationStack[OperationStackTop] do
begin
case CS of
                  '+'     : Operation := Plus;
                  '–'     : Operation := Minus;
                  '*'     : Operation := Mult;
        '/'       : Operation := Division
 end;  Priority := Prio
end
end;
```

И тогда окончательно получаем:

```
DYADIC OPERATION : Push Dyadic Operation, ( '+' ; '–' ; '*' ; '/') .
```

В свою очередь, правило (4) для нетерминала FACTOR можно преобразовать следующим образом:

```
FACTOR = NUMBER, Push Operand;
         Push Open Par,
                  '(', EXPRESSION , Unload And Discard Open Par, ')';
         Push Monadic Plus , '+' , FACTOR ;
         Push Monadic Minus , '–' , FACTOR
       =      ( Push Monadic Plus , '+' ; Push Monadic Minus , '–' )*,
                  ( NUMBER , Push Operand ;
              Push Open Par,
                  '(', EXPRESSION , Unload And Discard Open Par, ')' )
       = ( MONADIC OPERATION )*, OPERAND.
```

Здесь введены вспомогательные понятия: OPERAND и MONADIC OPERATION при помощи следующих правил:

```
MONADIC OPERATION : Push Monadic Operation , ( '+' ; '–' ) .
OPERAND : NUMBER , Push Operand ;
          Push Open Par,
              '(', EXPRESSION , Unload And Discard Open Par, ')'.
```

Новая унифицированная семантика обработки унарных операций Push Monadic Operation имеет следующую реализацию:

```
procedure PushMonadicOperation; {Помещает унарную операцию
                                 в магазин операций}
 begin Inc ( OperationStackTop );
   with OperationStack[OperationStackTop] do
   begin
     case CurrentSymbol of
          '+' : Operation := MonadicPlus;
          '–' : Operation := MonadicMinus
     end;
     Priority := 3
   end
end;
```

Подставляя в правило для нетерминала EXPRESSION вместо нетерминала FACTOR только что полученное выражение, получаем следующее правило:

```
EXPRESSION : ((MONADIC OPERATION)*, OPERAND) # DYADIC OPERATION.
```

Обратимся теперь к группе правил, определяющих числа:

```
NUMBER = INTEGER NUMBER , Set Integer Number ;
         REAL  NUMBER , Set Real Number
       = INTEGER NUMBER , Set Integer Number ;
         (FRACTION , Init Real with Fraction ;
```

```
    FRACTION , Init Real with Fraction ,
      EXPONENT PART , Append Exponent Part ;
    INTEGER NUMBER , Init Real with Integer ,
      EXPONENT PART, Append Exponent Part ) , SetRealNumber
= INTEGER NUMBER , Set Integer Number ;
    (FRACTION, Init Real with Fraction ,
        [EXPONENT PART, Append Exponent Part];
     INTEGER NUMBER , Init Real with Integer ,
       EXPONENT PART , Append Exponent Part ) , SetRealNumber
= INTEGER NUMBER ,
      ( Set Integer Number ;
        Init Real with Integer, EXPONENT PART ,
              Append Exponent Part, Set Real Number) ;
    FRACTION, Init Real with Fraction ,
      [EXPONENT PART , Append Exponent Part] , SetRealNumber
= INTEGER NUMBER ,
      (Set Integer Number ;
        (Init Real with Integer, EXPONENT PART,
          Append Exponent Part ;
         INTEGER NUMBER, InitFraction ,
          FRACTIONAL PART , AppendFractionalPart ,
          Init Real with Fraction, [EXPONENT PART ,
          Append Exponent Part]) , SetRealNumber)
= INTEGER NUMBER,
              (Set Integer Number ;
        (Init Real with Integer, EXPONENT PART , Append Exponent Part;
         InitFraction, FRACTIONAL PART , AppendFractionalPart ,
          Init Real with Fraction , [EXPONENT PART ,
          Append Exponent Part]) , SetRealNumber) .
```

На этом шаге преобразований становится ясно, что если бы семантики были ориентированы непосредственно на сборку вещественного числа (т.е. элемента операционной среды Number), а не промежуточных значений (RealNumber, IntegerNumber, Fraction), то выражение для понятия NUMBER можно было бы значительно упростить. Действительно, последнее выражение для понятия NUMBER с учетом переориентации семантик эквивалентно следующему:

```
NUMBER = INTEGER NUMBER ,
          ( Init Number with Integer ;
            Init Number with Integer , EXPONENT PART ,
                  Append Exponent Part ;
            Init Number with Integer , FRACTIONAL PART ,
                  AppendFractionalPart ,
                        [EXPONENT PART , Append Exponent Part])
       = INTEGER NUMBER , Init Number with Integer,
          [ EXPONENT PART , Append Exponent Part ;
        FRACTIONAL PART , AppendFractionalPart ,
          [EXPONENT PART , Append Exponent Part]]
       = INTEGER NUMBER , Init Number with Integer ,
          [FRACTIONAL PART , AppendFractionalPart] ,
          [EXPONENT PART , Append Exponent Part] .
```

Окончательно получаем правило

```
NUMBER : INTEGER NUMBER, Init Number with Integer,
                [FRACTIONAL PART, AppendFractionalPart],
                [EXPONENT PART, Append Exponent Part].
```

Семантические символы, используемые в нем, имеют следующую интерпретацию:

```
procedure Init Number with Integer;
begin Number := IntegerNumber end;
procedure AppendFractionalPart;
begin Number := Number + FractionalPart end;
procedure AppendExponentPart;
begin Number := Number * ExponentPart end;
```

Преобразуем теперь правило для INTEGER NUMBER:

```
INTEGER NUMBER  = DIGIT, Init Integer Number;
                = INTEGER NUMBER , DIGIT, Append Digit
                = DIGIT, Init Integer Number, (DIGIT, Append Digit)*
                = Init Integer, (DIGIT, Append Digit)+
                = Init Integer, (Set Digit, 'd', Append Digit)+.
```

В результате получаем правило

INTEGER NUMBER : Init Integer, (Set Digit, 'd', Append Digit)+.

Здесь новые семантики Init Integer и Set Digit имеют следующую реализацию:

```
procedure InitInteger;
begin IntegerNumber := 0; DigitsNmb := 0 end;
procedure SetDigit;
begin Digit := Ord(CurrentSymbol) – Ord ('0') end;
```

Правило для FRACTIONAL PART не нуждается в преобразованиях. Остается преобразовать лишь правило для EXPONENT PART:

```
EXPONENT PART = 'E', EXPONENT, Set Exponent Part
              = 'E', (INTEGER NUMBER, Set UnsignedExponent ;
                      SIGN, INTEGER NUMBER, Set Signed Exponent),
                        Set Exponent Part
              = 'E', [SIGN], INTEGER NUMBER,
                        Set Exponent, Set Exponent Part
              = 'E', [Set Positive, '+'; Set Negative, '–'],
                        INTEGER NUMBER , Set Exponent,
                        Set Exponent Part
              = 'E', Set Sign, (['+']; '–'), INTEGER NUMBER,
                        Set Exponent, Set Exponent Part.
```

Заметим, что при обработке порядка можно обойтись без переменной ExponentPart и обслуживающей ее семантики SetExponentPart, если выражение для нее из этой семантики подставить в последнее определение семантики AppendExponentPart.

Тогда получим:

```
EXPONENT PART : 'E' , Set Sign , (['+']; '–') , INTEGER NUMBER , SetExponent .
```

в предположении, что в правиле для понятия NUMBER используется следующая интерпретация семантического символа Append Exponent Part:

```
procedure AppendExponentPart;
begin
 if Exponent >= 0
 then Number := Number * Power (10, Exponent)
 else  Number := Number / Power (10, – Exponent)
end;
```

Семантики же Set Sign и Set Exponent имеют следующую реализацию:

```
procedure SetSign;
begin if CurrentSymbol = '–' then Sign := –1 else Sign := +1 end;
procedure SetExponent; begin Exponent := IntegerNumber * Sign end;
```

Заметим, что из правой части правила для нетерминала NUMBER посредством подстановок можно исключить нетерминалы INTEGER NUMBER, FRACTIONAL PART, EXPONENT PART, как и сам нетерминал NUMBER исключить из правой части правила, определяющего нетерминал OPERAND, а OPERAND, в свою очередь, — из правой части правила для конструкции EXPRESSION. Однако эти подстановки можно не производить вручную[110]. Они будут выполнены системой SYNTAX, если перевести эти символы из словаря нетерминалов в словарь вспомогательных понятий, сохраняя для них полученные правила.

Окончательная версия грамматики калькулятора, пригодная для построения процессора, реализующего интерпретацию арифметических выражений, приведена в разд. 1.3. Она отличается от грамматики, полученной в данном разделе, лишь тем, что в ней слегка сокращены семантические символы путем заключения части обозначений в скобки комментариев '{' и '}', и вместо семантического символа Init Number with Integer используется символ Set Integer Part. Кроме того, в реализацию семантики Set Digit внесены небольшие усовершенствования.

6.3. ГЕНЕРАЦИЯ ДИАГНОСТИЧЕСКИХ СООБЩЕНИЙ ОБ ОШИБКАХ

Использование вспомогательных понятий, кроме упомянутой автоматизации подстановок, способствует сохранению информации о первоначальной синтаксической структуре входного языка. Это позволяет генерировать диагностические сообщения об ошибках в терминах исходной структуры, более понятных пользователям программы синтаксически управляемой обработки данных, чем терминология рабочей грамматики.

[110] Их и не следует производить вручную, так как в этом случае будет потеряна структурная информация об исчезающих в результате этих подстановок конструкциях, а она могла бы использоваться в диагностических сообщениях об ошибках на входе. При автоматических подстановках эта информация сохраняется. Этот вопрос подробно обсуждается в разд. 6.3.

В самом деле, основной целью эквивалентных преобразований грамматики является ее максимальная регуляризация, фактически сводящаяся к исключению из грамматики как можно большего числа нетерминалов. Но именно нетерминалы несут всю структурную информацию о входном языке. Исключая нетерминалы, мы теряем возможность формулировать диагностические сообщения об ошибках в терминах первоначальной структуры входного языка. Например, из грамматики CALC могли бы быть исключены все нетерминалы, кроме двух: PROGRAM и EXPRESSION. Однако сохранение в качестве вспомогательных понятий символов NUMBER (ЧИСЛО), INTEGER NUMBER (ЦЕЛОЕ), FRACTIONAL PART (ДРОБНАЯ ЧАСТЬ), EXPONENT PART (ПОРЯДОК), DIGIT (ЦИФРА), MONADIC OPERATION (УНАРНАЯ ОПЕРАЦИЯ), DYADIC OPERATION (БИНАРНАЯ ОПЕРАЦИЯ) и OPERAND (ОПЕРАНД) с соответствующими синонимами (терминами, указанными в скобках) обеспечивает достаточный термино-логический базис для формирования диагностических сообщений об ошибках. Это достигается тем, что при построении управляющей граф-схемы производятся своего рода макроподстановки графов, представляющих правила для вспомогательных понятий вместо их использующих вхождений. При этом такие вставки маркируются именами соответствующих вспомогательных понятий. По маркам однозначно восстанавливается первоначальная синтакси-ческая структура конструкций, используемая в диагностических сообщениях[111].

В том, что подбор синонимов вспомогательных понятий грамматики CALC действительно удачен, можно убедиться, если сопоставить приводимые далее тексты диагностических сообщений, сгенерированных системой SYNTAX с управляющей таблицей калькулятора из разд. 1.5.

Диагностические сообщения CALC.

```
    В состоянии 1:
В конструкции ПРОГРАММА ожидается ВЫРАЖЕНИЕ
    В состоянии 2:
В конструкции ОПЕРАНД после компоненты '('
                              ожидается ВЫРАЖЕНИЕ
    В состоянии 3:
В конструкции ВЫРАЖЕНИЕ после компоненты УНАРНАЯ ОПЕРАЦИЯ
                              ожидается УНАРНАЯ ОПЕРАЦИЯ или ОПЕРАНД
    В состоянии 4:
В конструкции ВЫРАЖЕНИЕ после компоненты УНАРНАЯ ОПЕРАЦИЯ
                              ожидается УНАРНАЯ ОПЕРАЦИЯ или ОПЕРАНД
    В состоянии 5:
В конструкции ВЫРАЖЕНИЕ после компоненты ОПЕРАНД
                              ожидается БИНАРНАЯ ОПЕРАЦИЯ
В конструкции ЦЕЛОЕ после компоненты ЦИФРА ожидается ЦИФРА
В конструкции ЧИСЛО после компоненты ЦЕЛОЕ
                              ожидается ДРОБНАЯ ЧАСТЬ или ПОРЯДОК
```

[111] Разумеется, леворекурсивная структура, исключаемая посредством ее замены итеративной, не поддается восстановлению.

В состоянии 6:

В конструкции ДРОБНАЯ ЧАСТЬ после компоненты '.'
ожидается ЦЕЛОЕ

В состоянии 7:

В конструкции ПОРЯДОК после компоненты 'E'
ожидается '+' или ЦЕЛОЕ или '–'

В состоянии 8:

В конструкции ВЫРАЖЕНИЕ после компоненты БИНАРНАЯ ОПЕРАЦИЯ
ожидается УНАРНАЯ ОПЕРАЦИЯ или ОПЕРАНД

В состоянии 9:

В конструкции ВЫРАЖЕНИЕ после компоненты БИНАРНАЯ ОПЕРАЦИЯ
ожидается УНАРНАЯ ОПЕРАЦИЯ или ОПЕРАНД

В состоянии 10:

В конструкции ВЫРАЖЕНИЕ после компоненты БИНАРНАЯ ОПЕРАЦИЯ
ожидается УНАРНАЯ ОПЕРАЦИЯ или ОПЕРАНД

В состоянии 11:

В конструкции ВЫРАЖЕНИЕ после компоненты БИНАРНАЯ ОПЕРАЦИЯ
ожидается УНАРНАЯ ОПЕРАЦИЯ или ОПЕРАНД

В состоянии 12:

В конструкции ВЫРАЖЕНИЕ после компоненты ОПЕРАНД
ожидается БИНАРНАЯ ОПЕРАЦИЯ

В конструкции ЦЕЛОЕ после компоненты ЦИФРА
ожидается ЦИФРА

В конструкции ЧИСЛО после компоненты ДРОБНАЯ ЧАСТЬ
ожидается ПОРЯДОК

В состоянии 13:

В конструкции ПОРЯДОК после компоненты '+'
ожидается ЦЕЛОЕ

В состоянии 14:

В конструкции ПОРЯДОК после компоненты '–'
ожидается ЦЕЛОЕ

В состоянии 15:

В конструкции ВЫРАЖЕНИЕ после компоненты ОПЕРАНД
ожидается БИНАРНАЯ ОПЕРАЦИЯ

В конструкции ЦЕЛОЕ после компоненты ЦИФРА ожидается ЦИФРА

В состоянии 16:

В конструкции ПРОГРАММА после компоненты ВЫРАЖЕНИЕ
ожидается 'EOF'

В состоянии 17:

В конструкции ОПЕРАНД после компоненты ВЫРАЖЕНИЕ ожидается ')'

В состоянии 18: {Никаких диагностик}

В состоянии 19:

В конструкции ВЫРАЖЕНИЕ после компоненты ОПЕРАНД
ожидается БИНАРНАЯ ОПЕРАЦИЯ

Диагностические сообщения соотнесены с состояниями, в которых обнаруживаются ошибки[112]. Заметим, что для состояния 18, являющегося конечным, никакой диагностики не указано. Но это не означает, что всегда так: просто в данной конкретной ситуации для конечного состояния не существует никакой диагностической информации.

6.4. ИСПОЛЬЗОВАНИЕ РЕЗОЛЬВЕРОВ В ТРАНСЛЯЦИОННЫХ ГРАММАТИКАХ

До сих пор рассматривались примеры трансляционных грамматик, в которых резольверы не использовались. Но часто бывает необходимо определять трансляцию в зависимости от контекста. Такая ситуация возникает, например, при обработке описателей в языке программирования Алгол 68.

Описатели входят в состав некоторых конструкций и используются для спецификации видов. При этом одни конструкции используют фактические описатели (например, описания переменных, генераторы), другие — виртуальные (например, описатели имен), а третьи — формальные описатели (например, описания тождеств). Различие в описателях разного типа текстуально заметным образом сказывается лишь в описателях массивов. Во-первых, в фактических описателях массивов требуется указывать строгие границы индексов, в качестве которых можно использовать любые основы целого вида. Во-вторых, символ **flex** может быть первым символом лишь в фактических и виртуальных описателях. Например, в качестве фактических описателей массивов могли бы использоваться описатели `[5,10]`**int** или **flex**`[1:0]`**char**; как виртуальные: `[,]`**int** и **flex**`[]`**char**; в роли формальных: `[,]`**int** или `[]`**char**. С другой стороны, описатель может входить в состав другого описателя и при этом его тип, в общем случае, зависит от типа описателя, его объемлющего. Общие правила этой зависимости таковы:

- Описатели видов полей структуры и элементов массива имеют тот же тип, что и сам описатель структуры или массива.
- В описателе имени за символом **ref** всегда следует виртуальный описатель.
- Описатели параметров процедуры и ее результата, а также описатели объединенных видов — всегда формальные.

Например, локальный генератор

loc[5]**struct**([2][4,4]**int** *a*, **ref flex**[]**char** *b*, **proc**[]**real** *c*)

имеет следующий состав описателей:

---

[112] Строго говоря, ошибки обнаруживаются только в неподавляемых состояниях. Но подавляемые состояния, проходимые при процессировании, запоминаются в специальном магазине. В момент обнаружения ошибки выдаются диагностические сообщения, соотнесенные с состоянием, в котором эта ошибка обнаружена, а также с подавляемыми состояниями из магазина.

фактические : [5]**struct**([2][4,4]**int** *a*, **ref flex**[ ]**char** *b*, **proc**[ ]**real** *c*)
struct([2][4,4]**int** *a*, **ref flex**[ ]**char** *b*, **proc**[ ]**real** *c*)
[2][4,4]**int**
[4,4]**int**
int
ref flex[ ]**char**
виртуальные : **flex**[ ]**char**
char
фактический : **proc**[ ]**real**
формальные : [ ]**real**
real

Как видно из приведенной далее грамматики Gener, описывающей генераторы Алгола 68, структура описателей рекурсивна. Поэтому для запоминания типа текущего описателя на время анализа составляющего подописателя потребуется использование магазина с тем, чтобы после завершения разбора этого подописателя можно было восстановить тип текущего описателя, извлекая его из магазина.

Таким образом, при синтаксическом анализе только что приведенного генератора после того, как просканирован символ **loc**, должен быть установлен *фактический* тип ожидаемого далее описателя. Затем при получении символа **ref** текущий тип описателя (*фактический*) должен быть помещен в магазин, а в качестве нового текущего типа установлен *виртуальный*.

После завершения анализа описателя поля *‘b’* из магазина извлекается значение *фактический*, которое становится текущим типом до момента получения символа **proc**, с которого начинается описатель вида поля *‘c’* структуры. С этого момента текущим типом становится *формальный*, а прежний *(фактический)* попадает в магазин. После завершения анализа описателя вида поля *‘c’* из магазина извлекается значение *фактический*, которое далее используется в качестве текущего типа. Эти изменения текущего типа описателей обеспечиваются семантиками Set Actual, Set Formal, Set Virtual, а также Init и Reset.

С другой стороны, текущий тип описателя должен тестироваться при поступлении символа **flex** — он должен быть фактическим или виртуальным, и при получении основы в качестве строгой границы — он должен быть фактическим. Именно с этой целью в соответствующих позициях правил грамматики, определяющих понятия ROWS OF MODE DECLARATOR и ROWER, используются резольверы Is VirAct и Is Actual.

Приведем пример трансляционной грамматики, определяющей генераторы Алгола 68, с тем упрощением, что границы индексов будут представлены основами, в качестве которых могут использоваться лишь целые числа. Применяемая в ней лексика определяется грамматикой GenerLex. Она будет описана в следующей главе.

Gener -- генераторы Алгола 68 (модель).

LEXICS: GenerLex

SYNTAX

Nonterminals: GENERATOR (ГЕНЕРАТОР),
DECLARER (ОПИСАТЕЛЬ).

Auxiliary notions :
DECLARATOR (ОПРЕДЕЛИТЕЛЬ),
REFERENCE TO MODE DECLARATOR (ОПРЕДЕЛИТЕЛЬ_ИМЕНИ),
STRUCTURED WITH FIELDS DECLARATOR (ОПРЕДЕЛИТЕЛЬ_СТРУКТУРЫ),
ROWS OF MODE ECLARATOR (ОПРЕДЕЛИТЕЛЬ_МАССИВА),
PROCEDURE DECLARATOR (ОПРЕДЕЛИТЕЛЬ_ПРОЦЕДУРЫ),
UNION OF MODE DECLARATOR (ОПРЕДЕЛИТЕЛЬ_ОБЪЕДИНЕНИЯ),
PORTRAYER (СПЕЦИФИКАТОР_ПОЛЕЙ),
ROWER (ИНДЕКСАТОР),
UNIT (ОСНОВА),
ACTUAL DECLARER (ФАКТИЧЕСКИЙ_ОПИСАТЕЛЬ),
FORMAL DECLARER (ФОРМАЛЬНЫЙ_ОПИСАТЕЛЬ),
VIRTUAL DECLARER (ВИРТУАЛЬНЫЙ_ОПИСАТЕЛЬ).

Forward pass semantics : Init, Set Actual, Set Formal, Set Virtual, Reset, Complete.

Forward pass resolvers : Is VIRACT, Is Actual, Is Not Actual.

GENERATOR : Init, Set Actual, ('.loc' ; '.heap'), DECLARER, Complete.

DECLARER : 'ModeIndication'; DECLARATOR.

ACTUAL DECLARER : DECLARER.

FORMAL DECLARER: DECLARER.

VIRTUAL DECLARER: DECLARER.

DECLARATOR: REFRENCE TO MODE DECLARATOR;
STRUCTURED WITH FIELDS DECLARATOR;
ROWS OF MODE DECLARATOR;
PROCEDURE DECLARATOR;
UNION OF MODE DECLARATOR.

REFERENCE TO MODE DECLARATOR : Set Virtual, '.ref',
VIRTUAL DECLARER, Reset.

STRUCTURED WITH FIELDS DECLARATOR : '.struct', '(', PORTRAYER, ')'.

PORTRAYER : (DECLARER , 'tag' # ',') # ','.

ROWS OF MODE DECLARATOR : [Is VirAct, '.flex'], ROWER, DECLARER.

ROWER : '[', (Is Actual, [UNIT, ':'], UNIT; Is Not Actual , [':']) # ',', ']'.

UNIT : 'integer'.

PROCEDURE DECLARATOR: Set Formal , '.proc',
['(', FORMAL DECLARER # ',' , ')'],
FORMAL DECLARER, Reset.

UNION OF MODE DECLARATOR: Set Formal, '.union',
'(', FORMAL DECLARER # ',' , ')', Reset.

ENVIRONMENT

```
const n = 10;
type TSort = (Actual, Formal, Virtual);
     TLevel = 0..n; TIndex = 1..n;
     TStack = array [1..n] of TSORT;
var Stack : TStack; t : TLevel;
```

```
IMPLEMENTATION
                {Резольверы}
function IsVirAct: Boolean; { Выдает true, если текущий тип описателя
                                виртуальный или фактический, иначе — false}
begin IsVirAct := (Stack [t] = Virtual) or (Stack[t] = Actual) end;
function IsActual: Boolean; {Выдает true, если текущий тип описателя
                                фактический, иначе — false}}
begin IsActual := Stack [t]= Actual end;
function IsNotActual : Boolean; {Выдает true, если текущий тип описателя
                                    формальный или виртуальный, иначе — false }
begin IsNotActual := Not IsActual end;
                {Семантики}
procedure Init; {Опустошает магазин типов описателей}
begin t := 0 end;
procedure Reset; {Устанавливает тип объемлющего описателя}
begin Dec (t) end;
procedure SetActual; {Устанавливает фактический тип текущего описателя}
begin Inc (t); Stack[t]:= Actual end;
procedure SetFormal; {Устанавливает формальный тип текущего описателя}
begin Inc (t); Stack[t]:= Formal end;
procedure SetVirtual; {Устанавливает виртуальный тип текущего описателя }
begin Inc (t); Stack[t]:= Virtual end;
procedure Complete; {Выдает завершающее сообщение в окно результатов}
begin
NewLine;
PrintString ('Анализ генератора завершен успешно!')
end;
```

Контекстные символы существенно расширяют класс трансляций, которые можно задавать на базе бесконтекстных RBNF-грамматик. Используя резольверы и семантики, можно задавать контекстно зависимые трансляции.

Например, с их помощью можно определить видозависимые особенности синтаксиса, такие, как условия идентификации и согласованность видов подконструкций в составе данной конструкции.

Приведенный пример, хотя и демонстрирует использование контекстных символов, все же задает по существу бесконтекстную трансляцию. Вполне возможно построить RBNF-грамматику без контекстных символов, задающую ту же самую трансляцию[113]. Однако написать такую грамматику было бы сложнее, и она была бы значительно более громоздкой, поскольку для каждого описателя вида массива и структуры пришлось бы написать по три правила: одно — для фактического, другое — для формального, а третье — для виртуального описателя.

[113] Эта трансляция фактически состоит в распознавании описателей, единственный результат которого — выяснение, действительно ли входная цепочка является синтаксически правильным генератором.

Глава 7

МНОГОПРОЦЕССОРНАЯ ОБРАБОТКА

7.1. ВЗАИМОДЕЙСТВИЕ МЕЖДУ ЯЗЫКОВЫМИ ПРОЦЕССОРАМИ

На практике синтаксически управляемая обработка данных зачастую распределяется между несколькими процессорами одного или разных типов (конечными или сплайновыми, простыми или челночными). Взаимодействие между процессорами, обрабатывающими один входной поток, может организовываться двумя следующими способами:

процессор используется в качестве *сканера*, поставляющего лексемы другому процессору;

процессор используется как *семантическая* (*синтаксически управляемая*) *процедура* для другого процессора.

Процессор - сканер. Использование конечного процессора в качестве сканера типично для анализаторов языков программирования. Задача анализатора — выяснять конструкционный состав входной программы. Каждая конструкция представляется как цепочка терминальных символов, причем они имеют конкретное представление в виде цепочек литер. Таким образом, прежде чем станет возможным синтаксический анализ входной программы в некоторой грамматике, необходимо перевести ее из литерного представления в терминальное.

Распознавание терминалов — задача сканера, в качестве которого обычно используется некоторый конечный процессор. Этот конечный процессор нуждается в том, чтобы на его вход поступали литеры не сами по себе, а с соответствующими классифицирующими признаками, такими, как буква, цифра, спецзнак и т.п. Эти классифицирующие признаки к литерам добавляет *транслитератор*. Раздел спецификации микролексики сканера, начинающийся с ключевого слова **MICROLEXICS**, как раз и служит для определения микролексических классов, по которым транслитератор классифицирует входные литеры.

Входной поток любого процессора состоит из *лексем* — записей (структур) из трех полей: LC — номера лексического класса, LN — номера элемента в данном лексическом классе и LR — литерного представления элемента данного лексического класса.

Лексический класс — это множество входов, на которые управляющий процессор реагирует одинаково: поле LC есть просто код, который управляю-

щий процессор использует как ключ для доступа к соответствующему управляющему элементу. Два другие поля лексемы несут информацию, характеризующую конкретный элемент данного лексического класса. Поле LN можно использовать как индекс для доступа к атрибутам конкретного лексического элемента. При желании по его значению в некоторой семантике можно выбирать соответствующий вариант обработки. Поле LR в некоторых случаях также может быть использовано в семантиках или резольверах, например для формирования сообщения о недопустимости соответствующего лексического элемента на входе процессора или дополнительного анализа литерного представления лексемы.

Для определения состава лексических классов, по которым транслитератор классифицирует входные литеры, в грамматическом модуле имеется раздел микролексики, открываемый ключевым словом **MICROLEXICS**[114].

Транслитератор выдает значения LC, согласованные с порядковыми номерами терминалов, используемых в правилах грамматики в разделе SYNTAX того же грамматического модуля (TSL-спецификации). Их нумерация, начинающаяся с 1, определяется явно списком терминалов или, когда он не задан, неявно, текстуальным порядком первых вхождений терминалов в правила грамматики.

Два специальных значения LC зафиксировано для особых случаев: значение 0 приписано псевдомикролексеме **Eof** (логический конец файла), и значение –1 приписывается любой "незаконной" микролексеме, т.е. литере, не включенной ни в один из микролексических классов.

Идентификаторы функций LC, LN и LR могут использоваться в описании интерпретации семантических и резольверных символов процессора для доступа к соответствующим компонентам текущей, т.е. последней по времени принятия, входной лексемы.

Если некоторый процессор используется в роли сканера, то его задача из нескольких его собственных входных лексем сформировать одну выходную, к компонентам которых он может обращаться по именам LC, LN и LR. Для этого его семантикам, формирующим выходные лексемы, доступны стандартные имена OutLC, OutLN и OutLR. При этом номер лексического класса выходной лексемы следует брать из списка лексических классов в листинге управляющей граф-схемы, построенной по грамматике, использующей эту лексику как свои терминалы. Например, семантики из приводимой далее грамматики GenerLex, формирующие лексемы для процессора генераторов Gener, используют номера лексических классов, которые определены при построении управляющей граф-схемы по грамматике Gener. Разумеется, как всякий сканер, GenerLex игнорирует пробелы и комментарии (в фигурных скобках).

[114] Подробнее об этом см. в гл. 8.

GenerLex — лексика генераторов Алгола 68.

MICROLEXICS

LEXICAL CLASSES: '[', ']', '(', ')', ':', ',', '.', Letter, Digit, Blind, Escaped Symbols.

Letter : 'a'..'z'.

Digit : '0'..'9'.

Blind : #32, #9[115].

Escaped Symbols: #13,#10[116].

SYNTAX

Nonterminals : Lexeme.

Auxiliary notions : Integer, Tag, Bold Tag, Spec Symbol, Comment.

Terminals : 'Digit', '{', '}', 'Letter', '.', 'Blind', '[', ']', '(', ')', ',', ':'.

Forward pass semantics: Set Blind Visible, Set Blind Invisible, Set Mode Indication, Set Left Bracket, Set Right Bracket, Set Left Parenthesis, Set Right Parenthesis, Set Comma, Set Colon, Init Lex, Make Lex, Init Integer, Init Tag, Append, SetComment, Restore.

Forward pass resolvers : Is Key Word, Is Mode Indication.

Lexeme: (Comment ; 'Blind')*, Init Lex,
(Integer ; Tag ; Bold Tag ; Spec Symbol),
Make Lex, Comment*.

Comment: SetComment, '{', Restore, '}'.

Integer: Init Integer, (Append, 'Digit')+.

Tag: Init Tag, 'Letter', (Append, ('Letter' ; 'Digit'))*.

Bold Tag: Set Blind Visible, '.', Tag,
(Is Key Word ; Is Mode Indication, Set Mode Indication),
['Blind'], Set Blind Invisible.

Spec Symbol: Set Left Bracket, '[' ; Set Right Bracket, ']' ;
Set Left Parenthesis, '(' ; Set Right Parenthesis, ')' ;
Set Comma, ',' ;
Set Colon, ':'.

ENVIRONMENT

```
const  BlindItem = 6; RightBrace = 3; Illegal = –1; KeyWordNmb = 7;
       KeyWord: array [1..KeyWordNmb] of string =
            ('loc', 'heap', 'ref', 'flex','struct', 'proc', 'union');
var Repr : String; LexClass : Integer;
```

[115] Коды #32 и #9 обозначают пробел и табуляцию.

[116] Коды #13 и #10 обозначают перевод строки и возврат каретки.

```
IMPLEMENTATION

function IsKeyWord : Boolean; {Выдает true, если цепочка Repr, находится в
                               словаре ключевых слов KeyWord,
                               и false — в противном случае}
var i : 0..KeyWordNmb; AllScanned, Found : Boolean;
begin
  AllScanned := false; Found := false; i := 0;
  repeat
   Inc(i); AllScanned := i=KeyWordNmb;
   Found := KeyWord[i]=Repr;
   if Found then
   begin
      case i of
            1 {loc}    : LexClass := 1;
            2 {heap}   : LexClass := 2;
            3 {ref}    : LexClass := 4;
            4 {flex}   : LexClass :=10;
            5 {struct} : LexClass := 5;
            6 {proc}   : LexClass :=15;
            7 {union}  : LexClass :=16;
        end
   end
  until (Found or AllScanned);
  IsKeyWord := Found
end;

function IsModeIndication : Boolean; {Выдает true, если на входе не ключевое
                                      слово, и false — в противном случае}
begin IsModeIndication := Not IsKeyWord end;

procedure SetBlindVisible; {Делает символы пробела и табуляции видимыми}
begin SetVisibleOn (BlindItem) end;

procedure SetBlindInVisible; {Делает символы пробела и табуляции невидимыми}
begin SetVisibleOff (BlindItem) end;

procedure SetModeIndication; {Подготавливает выходную лексему ModeIndication}
begin LexClass := 3; Repr := '.'+Repr end;

procedure SetComment; {Устанавливает режим комментария: делает все литеры,
                       кроме '}', невидимыми}
begin SetInvisible; SetVisibleOn(RightBrace) end;

procedure Restore; {Делает все литеры, кроме литер пробела и табуляции,
                    видимыми}
begin SetVisible; SetVisibleOff (BlindItem) end;
```

```
procedure Append; {Накапливает литерное представление лексемы}
begin Repr := Repr + LR end;

procedure InitLex; {Инициализирует формирование выходной лексемы}
begin Repr := ''; LexClass := Illegal end;

procedure InitInteger; {Подготавливает формирование выходной лексемы Integer}
begin  LexClass := 14;  Repr := '' end;

procedure MakeLex; {Завершает формирование выходной лексемы}
begin  OutLR := Repr; OutLN := 0; OutLC := LexClass; NewLine;
       PrintString (OutLR)
end;

procedure SetColon; {Подготавливает формирование выходной лексемы Colon}
begin LexClass := 12; Repr := ':' end;

procedure SetComma; {Подготавливает формирование выходной лексемы Comma}
begin LexClass := 9; Repr := ',' end;

procedure SetLeftBracket; {Подготавливает формирование выходной лексемы
                                    LeftBracket}
begin LexClass := 11; Repr := '[' end;

procedure  SetLeftParenthesis; {Подготавливает формирование выходной лексемы
                                    LeftParenthesis}
begin LexClass := 6;  Repr := '('  end;

procedure  SetRightBracket; {Подготавливает формирование выходной лексемы
                                    RightBracket}
begin LexClass := 13; Repr := ']' end;

procedure SetRightParenthesis; {Подготавливает формирование выходной лексемы
                                    RightParenthesis}
begin LexClass := 7; Repr := ')' end;

procedure InitTag; {Инициализирует сборку идентификатора (Tag)}
begin Repr := LR; LexClass := 8 end;
```

Следует пояснить, что ключевые слова и индикаторы видов в Алголе 68 выделяются полужирным шрифтом, который при машинном кодировании имитируется при помощи стробирования. В качестве стробирующего символа используется точка. Ее действие в качестве стробирующего символа распространяется на следующую непосредственно за ней букву и все последующие буквы и цифры до ближайшей литеры, не являющейся ни буквой, ни цифрой. Это учтено в спецификации сканера.

Итак, для анализа генераторов Алгола 68 мы используем комплекс, состоящий из транслитератора, сканера (конечного процессора) и анализатора (сплайнового процессора), схема взаимодействия между которыми изображена на рис. 7.1.

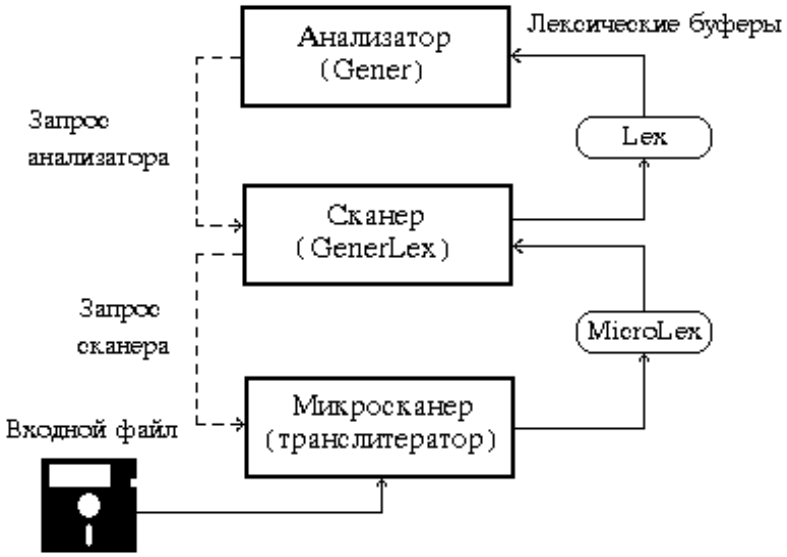


Рис. 7.1. Схема взаимодействия компонент
комплекса анализа генераторов.

Анализатор, сканер и транслитератор сообщаются между собой через лексические буферы. Когда процессор высшего уровня запрашивает очередную лексему, процессор низшего уровня поставляет ее в промежуточный лексический буфер. Она остается там до тех пор, пока первый процессор, приняв ее, не запросит следующую входную лексему. Такое взаимодействие происходит между каждой парой процессоров соседних уровней. Поставщиком микролексем для транслитератора служит входной поток литер.

Распределение задач анализа между несколькими языковыми процессорами улучшает технологические свойства SYNTAX-приложения и благотворно сказывается на точности диагностических сообщений об ошибках во входном тексте.

7.2. ДИНАМИЧЕСКОЕ УПРАВЛЕНИЕ ВИДИМОСТЬЮ ВХОДНЫХ ЛЕКСЕМ

На практике часто возникает необходимость сделать некоторые входные лексемы "невидимыми" для процессора. Это значит, что если текущая лексема невидима, то процессор, не обрабатывая ее, запрашивает следующую. Причем ситуация в отношении видимости входных лексем для данного процессора может изменяться динамически путем вызова семантик во время его работы.

Примерами, в которых такая потребность возникает, являются анализаторы языков программирования. В программе на языке программирования почти везде можно вставлять пробелы, табуляции и комментарии, не изменяя ее смысла. Но в некоторых местах программы пробелы могут быть значимы, например в строках. Сканер, распознающий строку и формирующий соответст-

вующую выходную лексему, получив на вход микролексему " (кавычки), должен воспринимать все последующие входные микролексемы, включая и микролексему 'пробел'. Получив же на своем входе повторно микролексему " (кавычки), он должен перестать ‘замечать’ пробелы. Или же, получив левую скобку комментария, сканер должен перестать видеть все последующие символы до того момента, как на его входе не появится закрывающая комментарий скобка.

В рассмотренном примере процессора GenerLex потребность переключать видимость пробелов, табуляций и комментариев, т.е. последовательностей любых допустимых символов, заключенных в фигурные скобки, возникает несколько раз. Написать правила грамматики, которые порождали бы почти всюду такие необязательные элементы языка, было бы весьма затруднительно. Чтобы облегчить решение этой проблемы для пользователя, в операционную среду каждого процессора автоматически включается логическая шкала ‘видимости’ входных символов. Элементы этой шкалы — логические значения — соответствуют входным лексическим классам. При создании процессора ее элементы автоматически инициализируются значениями **true** (входные лексемы всех классов видимы). Если *i*-й элемент этой шкалы имеет значение **true**, то входная лексема класса *i* видна и используется для определения соответствующего управляющего элемента. Если же *i*-й элемент имеет значение **false**, то процессор запрашивает очередную входную лексему, и ее видимость тестируется так же.

Семантикам и резольверам процессора, которые определяет пользователь, доступ к элементам этой шкалы открыт через следующие стандартные процедуры операционной среды любого данного процессора:

procedure SetVisibleOn (LC : Integer); {Включает обработку входной лексемы класса LC[117] , т. е. делает ее "видимой" для данного процессора}

procedure SetVisibleOff (LC : Integer); {Выключает обработку входной лексемы класса LC, т.е. делает ее "невидимой" для данного процессора}

procedure SetInvisible; {Делает невидимыми все входные лексемы процессора}

procedure SetVisible; {Делает видимыми все входные лексемы процессора}

procedure SaveVisibilityMode; {Сохраняет текущий режим видимости входных лексем процессора}

procedure RestoreVisibilityMode; {Восстанавливает предыдущий режим видимости входных лексем процессора}

Использование этих процедур можно наблюдать по ранее приведенной спецификации сканера для генераторов Алгола 68.

[117] Номер лексического класса LC определяется по списку терминалов, генерируемому при построении управляющей граф-схемы. Этот список можно увидеть в листинге.

7.3. АНАЛИЗ ГЕНЕРАТОРОВ АЛГОЛА 68

Далее приведем отчетную документацию по комплексу из двух процессоров и транслитератора для анализа генераторов Алгола 68. В нее входят листинги управляющих граф-схем и управляющих таблиц, полученные в подсистеме проектирования по двум грамматикам Gener (см. гл. 6) и GenerLex (см. разд. 7.1), а также протоколы пропуска трех тестовых вариантов генераторов, полученных в подсистеме процессирования, которые демонстрируют ход анализа одного правильного генератора Алгола 68 и диагностику ошибок в двух неправильных генераторах. В первом случае — это бесконтекстная ошибка, диагностируемая встроенным механизмом процессора, а во втором — контекстная, диагностируемая семантиками, определенными разработчиком.

Управляющая граф-схема Gener:

0	begin	GENERATOR
1	FS	Init
2	FS	SetActual
3	–<	6
4	T	'.loc'
5	→	7
6	T	'.heap'
7	{	ACTUAL DECLARER
8	N	DECLARER
9	{	ACTUAL DECLARER
10	FS	Complete
11	end	GENERATOR
12	begin	DECLARER
13	–<	16
14	T	'ModeIndication'
15	→	106
16	{	DECLARATOR
17	–<	27
18	{	REFERENCE TO MODE DECLARATOR
19	FS	SetVertual
20	T	'.ref'
21	{	VIRTUAL DECLARER
22	N	DECLARER
23	}	VIRTUAL DECLARER
24	FS	Reset
25	}	REFRENCE TO MODE DECLARATOR
26	→	105
27	–<	44
28	{	STRUCTURED WITH FIELDS DECLARATOR
29	T	'.struct'
30	T	'('
31	{	PORTRAYER
32	N	DECLARER
33	T	'tag'
34	–<	37
35	T	','
36	→	33
37	–<	40
38	T	','
39	→	32
40	}	PORTRAYER
41	T	')'
42	}	STRUCTURED WITH FIELDS DECLARATOR
43	→	105
44	–<	73
45	{	ROWS OF MODE DECLARATOR
46	–<	49
47	FR	IsVIRACT
48	T	'.flex'
49	{	ROWER
50	T	'['
51	–<	62
52	FR	IsActual
53	–<	58
54	{	UNIT
55	T	'integer'

56	}	UNIT
57	T	‘:’
58	{	UNIT
59	T	‘integer’
60	}	UNIT
61	→	65
62	FR	IsNotActual
63	–<	65
64	T	‘:’
65	–<	68
66	T	‘,’
67	→	51
68	T	‘]’
69	}	ROWER
70	N	DECLARER
71	}	ROWS OF MODE DECLARATOR
72	→	105
73	–<	92
74	{	PROCEDURE DECLARATOR
75	FS	SetFormal
76	T	‘.proc’
77	–<	86
78	T	‘(’
79	{	FORMAL DECLARER
80	N	DECLARER
81	}	FORMAL DECLARER
82	–<	85
83	T	‘,’
84	→	79
85	T	‘)’
86	{	FORMAL DECLARER
87	N	DECLARER
88	}	FORMAL DECLARER
89	FS	Reset
90	}	PROCEDURE DECLARATOR
91	→	105
92	{	UNION OF MODE DECLARATOR
93	FS	SetFormal
94	T	‘.union’
95	T	‘(’
96	{	FORMAL DECLARER
97	N	DECLARER
98	}	FORMAL DECLARER
99	–<	102
100	T	‘,’
101	→	96
102	T	‘)’
103	FS	Reset
104	}	UNION OF MODE DECLARATOR
105	}	DECLARATOR
106	end	DECLARER

Словарь терминалов:

1	.loc	5	.struct	9	,	13	]
2	.heap	6	(	10	.flex	14	integer
3	ModeIndication	7	)	11	[	15	.proc
4	.ref	8	tag	12	:	16	.union

Словарь нетерминалов:

	Acronym:	Synonym:
1	DECLARER	(ОПИСАТЕЛЬ)
2	GENERATOR	(ГЕНЕРАТОР)

Словарь вспомогательных понятий:

	Acronym:	Synonym:
1	ACTUAL DECLARER	(ФАКТИЧЕСКИЙ_ОПИСАТЕЛЬ)
2	DECLARATOR	(ОПРЕДЕЛИТЕЛЬ)
3	FORMAL DECLARER	(ФОРМАЛЬНЫЙ_ОПИСАТЕЛЬ)
4	PORTRAYER	(СПЕЦИФИКАТОР_ПОЛЕЙ)
5	PROCEDURE DECLARATOR	(ОПРЕДЕЛИТЕЛЬ_ПРОЦЕДУРЫ)
6	REFERENCE TO MODE DECLARATOR	(ОПРЕДЕЛИТЕЛЬ_ИМЕНИ)
7	ROWER	(ИНДЕКСАТОР)
8	ROWS OF MODE DECLARATOR	(ОПРЕДЕЛИТЕЛЬ_МАССИВА)
9	STRUCTURED WITH FIELDS DECLARATOR	(ОПРЕДЕЛИТЕЛЬ_СТРУКТУРЫ)
10	UNION OF MODE DECLARATOR	(ОПРЕДЕЛИТЕЛЬ_ОБЪЕДИНЕНИЯ)
11	UNIT	(ОСНОВА)
12	VIRTUAL DECLARER	(ВИРТУАЛЬНЫЙ_ОПИСАТЕЛЬ)

Словари контекстных символов:

	Forward Pass Semantics:			Forward Pass Resolvers:	
1	Complete	4	SetActual	1	IsActual
2	Init	5	SetFormal	2	IsNotActual
3	Reset	6	SetVirtual	3	IsVIRACT

Управляющая таблица прямого просмотра Gener. Сначала приведем таблицу управляющих элементов[118] (табл. 7.1), а затем таблицу возвратных состояний анализатора генераторов Алгола 68 (табл. 7.2).

Таблица 7.1

Вход	Резольвер	Семантика	Магазин	Состояние
1	**2**	**3**	**4**	**5**
Состояние 1={0}				
.loc		Init;SetActual		2
.heap		Init;SetActual		3
Состояния 2={4}, 3={6}				
ModeIndication			1	4
.ref		SetVirtual	1	5
.struct			1	6
.flex	IsVIRACT		1	7
[			1	8
.proc		SetFormal	1	9
.union		SetFormal	1	10

[118] Совмещенную с таблицей резольверов в форме дополнительной графы Резольвер в составе таблицы управляющих элементов.

Продолжение табл. 7.1

Состояния 4={14}, 24={70}, 32={41}				
ε				**Sup**
Состояние 5={20}				
ModeIndication			2	4
.ref		SetVirtual	2	5
.struct			2	6
.flex	IsVIRACT		2	7
[			2	8
.proc		SetFormal	2	9
.union		SetFormal	2	10
Состояние 6={29}				
.struct				11
Состояние 7={48}				
[				8
Состояния 8={50}, 12={66}				
,	IsNotActual			12
:	IsNotActual			13
]	IsNotActual			14
integer	IsActual			15
Состояние 9={76}				
ModeIndication			3	4
.ref		SetVirtual	3	5
.struct			3	6
(				16
.flex	IsVIRACT		3	7
[			3	8
.proc		SetFormal	3	9
.union		SetFormal	3	10
Состояние 10={29}				
(				17
Состояние 11={30}				
ModeIndication			4	4
.ref		SetVirtual	4	5
.struct			4	6
.flex	IsVIRACT		4	7
[			4	8
.proc		SetFormal	4	9
.union		SetFormal	4	10

Продолжение табл. 7.1

Состояния 13 = {64}, 19 = {59}				
,				12
]				14
Состояние 14 = {68}				
ModeIndication			5	4
.ref		SetVirtual	5	5
.struct			5	6
.flex	IsVIRACT		5	7
[			5	8
.proc		SetFormal	5	9
.union		SetFormal	5	10
Состояние 15 = {55 59}				
,				12
:				18
]				14
Состояния 16 = {78}, 29 = {83}				
ModeIndication			6	4
.ref		SetVirtual	6	5
.struct			6	6
.flex	IsVIRACT		6	7
[			6	8
.proc		SetFormal	6	9
.union		SetFormal	6	10
Состояния 17 = {95}, 31 = {100}				
ModeIndication			7	4
.ref		SetVirtual	7	5
.struct			7	6
.flex	IsVIRACT		7	7
[			7	8
.proc		SetFormal	7	9
.union		SetFormal	7	10
Состояние 18 = {57}				
integer		Complete		19
Состояние 20 = {8} (конечное)				
ε		Complete		**Stop**

Продолжение табл. 7.1

Состояния 21 = {22}, 22 = {87}, 30 = {102}				
ε		Reset		**Sup**
Состояние 23 = {32}				
tag				27
Состояние 25 = {80}				
)				28
,				29
Состояние 26 = {80}				
)				30
,				31
Состояние 27 = {33}				
)				32
,				33
Состояние 28 = {85}				
ModeIndication			3	4
.ref		SetVirtual	3	5
.struct			3	6
.flex	IsVIRACT		3	7
[			3	8
.proc		SetFormal	3	9
.union		SetFormal	3	10
Состояние 33 = {35 38}				
ModeIndication			4	4
.ref		SetVirtual	4	5
.struct			4	6
tag				27
.flex	IsVIRACT		4	7
[			4	8
.proc		SetFormal	4	9
.union		SetFormal	4	10

Таблица 7.2

Подавляемые состояния: 4, 21, 22, 24, 30, 32

Магазинный символ	Возвратное состояние
1	20
2	21
3	22
4	23
5	24
6	25
7	26

***Замечание* 1.** Одинаковые таблички, относящиеся к различным состояниям, ради краткости сведены под общие заголовки, содержащие характеристики соответствующих состояний. Характеристика состояния — некоторое множество вершин управляющей граф-схемы. Вершины заданы номерами представляющих их записей управляющей граф-схемы.

***Замечание* 2.** Таблица возвратных состояний (табл. 7.2) фактически состоит из шести совершено одинаковых подтаблиц для подавляемых состояний 4, 21, 22, 24, 30 и 32[119]. Это означает, что возвратные состояния не зависят от подавляемых. Они однозначно определяются магазинными символами. Как следствие этого обстоятельства, после оптимизации процессора Gener таблица возвратных состояний исчезает, а в управляющих элементах магазинные символы замещаются соответствующими возвратными состояниями, представленными их номерами со знаком минус — см. далее табл. 7.4. Размеры компонент управляющей таблицы Gener приведены в табл. 7.3. Большинство из них представляются динамическими массивами, состоящими из дескрипторов (Descr) и собственно элементов, размещаемых в пулах (Pool).

Таблица 7.3

Компонента		Элементов	Байт
Характеристики состояний:			
	Descr	33	132
	Pool	34	68
Лексический вход:	Descr	33	132
	Pool	90	540
Резольверный вход		11	44
Пул управляющих элементов		71	426
Семантические цепочки:	Descr	5	20
	Pool	6	12
Резольверные цепочки:	Descr	3	12
	Pool	3	6
Магазинные цепочки:	Descr	7	28
	Pool	7	14
Магазинные символы:	Descr	7	28
	Pool	7	14
Конечные состояния		1	2
Табл. возвр. состояний:	Descr	1	4
	Pool	7	28
			1510

Суммарный объем (в байтах) всех компонент таблицы позволит в дальнейшем судить о степени экономии памяти за счет оптимизации управляющих таблиц обсуждаемых процессоров.

[119] В них графа Резольверы опущена, поскольку в данном примере возвратные состояния не зависят от состояния операционной среды.

Оптимизация управляющей таблицы прямого просмотра процессора Gener производится путем замены состояний, входных и магазинных символов (в табл. 7.1, 7.2) соответствующими классами эквивалентности, которые указаны ниже,

— для состояний:

1:	{1}	7:	{8 12}	13:	{15}	19:	{23}
2:	{2 3}	8:	{9}	14:	{16 29}	20:	{25}
3:	{4 24 32}	9:	{10}	15:	{17 31}	21:	{26}
4:	{5 28}	10:	{11}	16:	{18}	22:	{27}
5:	{6}	11:	{13 19}	17:	{20}	23:	{33}
6:	{7}	12:	{14}	18:	{21 22 30}		

— для входных символов:

1:	{ .loc .heap }	6:	{) }	11:	{ : }
2:	{ ModeIndication }	7:	{ tag }	12:	{] }
3:	{ .ref }	8:	{ , }	13:	{ integer }
4:	{ .struct }	9:	{ .flex }	14:	{ .proc }
5:	{ (}	10:	{ [}	15:	{ .union }

и для магазинных символов:

1:	{1}	3:	{4}	5:	{6}
2:	{2 3}	4:	{5}	6:	{7}

В результате оптимизации процессора Gener получается управляющая таблица, которая состоит только из таблицы управляющих элементов[120] (табл. 7.4). Таблица возвратных состояний, как уже отмечалось, в результате оптимизации исключена.

Таблица 7.4

Вход	Резольвер	Семантика	Магазин	Состояние
1	**2**	**3**	**4**	**5**
Класс состояний 1 = {1}				
[.loc]		Init; SetActual		2
Класс состояний 2 = {2 3}				
[ModeIndication]			–17	3
[.ref]		SetVirtual	–17	4
[.struct]			–17	5
[.flex]	IsVIRACT		–17	6
[[]			–17	7
[.proc]		SetFormal	–17	8
[.union]		SetFormal	–17	9

[120] Совмещенной с таблицей резольверов.

Продолжение табл. 7.4

Класс состояний 3 = {4 24 32}				
ε				**Sup**
Класс состояний 4 = {5 28}				
[ModeIndication]			–18	3
[.ref]		SetVirtual	–18	4
[.struct]			–18	5
[.flex]	IsVIRACT		–18	6
[[]			–18	7
[.proc]		SetFormal	–18	8
[.union]		SetFormal	–18	9
Класс состояний 5 = {6}				
[(]				10
Класс состояний 6 = {7}				
[]]				7
Класс состояний 7 = {8 12}				
[,]	IsNotActual			7
[:]	IsNotActual			11
[]]	IsNotActual			12
[integer]	IsActual			13
Класс состояний 8 = {9}				
[ModeIndication]			–18	3
[.ref]		SetVirtual	–18	4
[.struct]			–18	5
[(]				14
[.flex]	IsVIRACT		–18	6
[[]			–18	7
[.proc]		SetFormal	–18	8
[.union]		SetFormal	–18	9
Класс состояний 9 = {10}				
[(]				15
Класс состояний 10 = {11}				
[ModeIndication]			–19	3
[.ref]		SetVirtual	–19	4
[.struct]			–19	5
[.flex]	IsVIRACT		–19	6
[[]			–19	7
[.proc]		SetFormal	–19	8
[.union]		SetFormal	–19	9

Продолжение табл. 7.4

Класс состояний 11 = {13 19}				
[,]				7
[]]				12
Класс состояний 12={14}				
[ModeIndication]			–3	3
[.ref]		SetVirtual	–3	4
[.struct]			–3	5
[.flex]	IsVIRACT		–3	6
[[]			–3	7
[.proc]		SetFormal	–3	8
[.union]		SetFormal	–3	9
[]]				7
Класс состояний 13 = {15}				
[,]				7
[:]				16
[]]				12
Класс состояний 14 = {16 29}				
[ModeIndication]			–20	3
[.ref]		SetVirtual	–20	4
[.struct]			–20	5
[.flex]	IsVIRACT		–20	6
[[]			–20	7
[.proc]		SetFormal	–20	8
[.union]		SetFormal	–20	9
Класс состояний 15 = {17 31}				
[ModeIndication]			–21	3
[.ref]		SetVirtual	–21	4
[.struct]			–21	5
[.flex]	IsVIRACT		–21	6
[[]			–21	7
[.proc]		SetFormal	–21	8
[.union]		SetFormal	–21	9
Класс состояний 16 = {18}				
[integer]		Complete		11
Класс состояний 17 = {20} (конечных)				
ε		Complete		**Stop**
Класс состояний 18 = {21 22 30}				
ε		Reset		**Sup**
Класс состояний 19={23}				
[tag]				22
Класс состояний 20={25}				
[)]				4
[,]				14

Продолжение табл. 7.4

Класс состояний 21={26}				
[)]				18
[,]				15
Класс состояний 22 = {27}				
[)]				3
[,]				23
Класс состояний 23 = {33}				
[ModeIndication]			–19	3
[.ref]		SetVirtual	–19	4
[.struct]			–19	5
[tag]				22
[.flex]	IsVIRACT		–19	6
[[]			–19	7
[.proc]		SetFormal	–19	8
[.union]		SetFormal	–19	9

В этой таблице отрицательные номера в графе Магазин представляют возвратные состояния[121].

Количественные характеристики компонент оптимизированной управляющей таблицы процессора Gener приведены в табл. 7.5.

Таблица 7.5

Компонента		Элементов	Байт
Классы состояний:	Descr	23	92
	Pool	33	66
Лексический вход:	Descr	33	92
	Pool	82	492
Резольверный вход		10	40
Пул управляющих элементов		59	354
Семантические цепочки:	Descr	5	20
	Pool	6	12
Резольверные цепочки:	Descr	3	12
	Pool	3	6
Магазинные цепочки:	Descr	6	24
	Pool	6	12
Классы магазинных символов:			
	Descr	6	24
	Pool	7	14
Классы конечных состояний		1	2
Табл. возвр. состояний:	Descr	0	0
	Pool	0	0
			1262

[121] В общем случае в результате оптимизации не все магазинные символы замещаются возвратными состояниями. Знак минус как раз и служит для их отличия от "обычных" магазинных символов.

Сравнение размеров управляющих таблиц неоптимизированного (см. табл. 7.3) и оптимизированного (см. табл. 7.5) вариантов процессора Gener показывает, что оптимизация дает в этом случае 16%-ную экономию памяти.

Диагностики оптимизированного процессора Gener. Генератор диагностических сообщений периода процессирования для оптимизированного процессора Gener дает следующий набор диагностик:

В классе состояний 1:

```
В конструкции ГЕНЕРАТОР ожидается '.loc', '.heap'
```

В классе состояний 2:

```
В конструкции ГЕНЕРАТОР после компоненты '.heap'
                        ожидается ФАКТИЧЕСКИЙ ОПИСАТЕЛЬ
В конструкции ГЕНЕРАТОР после компоненты '.loc'
                        ожидается ФАКТИЧЕСКИЙ ОПИСАТЕЛЬ
```

В классе состояний 3: {Никаких диагностик}

В классе состояний 4:

```
В конструкции ОПРЕДЕЛИТЕЛЬ ИМЕНИ
после компоненты '.ref' ожидается ВИРТУАЛЬНЫЙ ОПИСАТЕЛЬ
В конструкции ОПРЕДЕЛИТЕЛЬ ПРОЦЕДУРЫ
после компоненты ')' ожидается ФОРМАЛЬНЫЙ ОПИСАТЕЛЬ
```

В классе состояний 5:

```
В конструкции ОПРЕДЕЛИТЕЛЬ СТРУКТУРЫ
после компоненты '.struct' ожидается '('
```

В классе состояний 6:

```
В конструкции ОПРЕДЕЛИТЕЛЬ МАССИВА
после компоненты '.flex' ожидается ИНДЕКСАТОР
```

В классе состояний 7:

```
В конструкции ИНДЕКСАТОР после компоненты ',' ожидается ОСНОВА, ':', ']'
```

В классе состояний 8:

```
В конструкции ОПРЕДЕЛИТЕЛЬ ПРОЦЕДУРЫ
после компоненты '.proc' ожидается '(', ФОРМАЛЬНЫЙ ОПИСАТЕЛЬ
```

В классе состояний 9:

```
В конструкции ОПРЕДЕЛИТЕЛЬ ОБЪЕДИНЕНИЯ
после компонеты '.union' ожидается '('
```

В классе состояний 10:

```
В конструкции ОПРЕДЕЛИТЕЛЬ СТРУКТУРЫ
после компоненты '(' ожидается СПЕЦИФИКАТОР ПОЛЕЙ
```

В классе состояний 11:

```
В конструкции ИНДЕКСАТОР после компоненты ':' ожидается ',', ']'
В конструкции ИНДЕКСАТОР после компоненты ОСНОВА ожидается ',', ']'
```

В классе состояний 12:

```
В конструкции ОПРЕДЕЛИТЕЛЬ МАССИВА
после компоненты ИНДЕКСАТОР ожидается ОПИСАТЕЛЬ
```

В классе состояний 13:

В конструкции ИНДЕКСАТОР после компоненты ОСНОВА ожидается ',', ']'

В конструкции ИНДЕКСАТОР после компоненты ОСНОВА ожидается ':'

В классе состояний 14:

В конструкции ОПРЕДЕЛИТЕЛЬ ПРОЦЕДУРЫ
после компоненты '(' ожидается ФОРМАЛЬНЫЙ ОПИСАТЕЛЬ

В конструкции ОПРЕДЕЛИТЕЛЬ ПРОЦЕДУРЫ
после компоненты ',' ожидается ФОРМАЛЬНЫЙ ОПИСАТЕЛЬ

В классе состояний 15:

В конструкции ОПРЕДЕЛИТЕЛЬ ОБЪЕДИНЕНИЯ

после компоненты '(' ожидается ФОРМАЛЬНЫЙ ОПИСАТЕЛЬ

В конструкции ОПРЕДЕЛИТЕЛЬ ОБЪЕДИНЕНИЯ
после компоненты ',' ожидается ФОРМАЛЬНЫЙ ОПИСАТЕЛЬ

В классе состояний 16:

В конструкции ИНДЕКСАТОР после компоненты ':' ожидается ОСНОВА

В классах состояний 17 и 18: {Никаких диагностик}

В классе состояний 19:

В конструкции СПЕЦИФИКАТОР ПОЛЕЙ
после компоненты ОПИСАТЕЛЬ ожидается 'tag'

В классе состояний 20:

В конструкции ОПРЕДЕЛИТЕЛЬ ПРОЦЕДУРЫ
после компоненты ФОРМАЛЬНЫЙ ОПИСАТЕЛЬ ожидается ',', ')'

В классе состояний 21:

В конструкции ОПРЕДЕЛИТЕЛЬ ОБЪЕДИНЕНИЯ
после компоненты ФОРМАЛЬНЫЙ ОПИСАТЕЛЬ ожидается ',', ')'

В классе состояний 22:

В конструкции ОПРЕДЕЛИТЕЛЬ СТРУКТУРЫ
после компоненты СПЕЦИФИКАТОР ПОЛЕЙ ожидается ')'

В конструкции СПЕЦИФИКАТОР ПОЛЕЙ после компоненты 'tag'ожидается ','

В классе состояний 23:

В конструкции СПЕЦИФИКАТОР ПОЛЕЙ после компоненты ',' ожидается 'tag'

В конструкции СПЕЦИФИКАТОР ПОЛЕЙ
после компоненты ',' ожидается ОПИСАТЕЛЬ

Замечание. Как видим, к одному классу эквивалентных состояний может относиться несколько вариантов диагностических сообщений. Это связано с возможностью трактовки ошибки с точки зрения нескольких конструкций разного уровня вложенности или неоднозначностью распознавания конструкций одного уровня по их префиксам. Изредка, как, например, в классах состояний 3, 17 и 18, не существует никакой информации для формирования диагностик. Как исключение, диагностические сообщения, относящиеся к начальному классу состояний (первому), не содержат в своей рамке части "после …". Это и понятно, поскольку в нем еще только ожидается первый символ входной цепочки.

7.4. ЛЕКСИКА ГЕНЕРАТОРОВ

Продолжим обсуждение процессора GenerLex, используемого в качестве сканера для анализатора генераторов Алгола 68. Начнем с граф-схемы, построенной по грамматике GenerLex (см. разд. 7.1).

Управляющая граф-схема GenerLex:

0	begin	Lexeme	29	→	31
1	–<	12	30	T	'Digit'
2	–<	10	31	→	25
3	{	Comment	32	}	Tag
4	FS	SetComment	33	→	83
5	T	'{'	34	–<	59
6	FS	Restore	35	{	BoldTag
7	T	'}'	36	FS	SetBlindVisible
8	}	Comment	37	T	'.'
9	→	11	38	{	Tag
10	T	Blind	39	FS	InitTag
11	→	1	40	T	'Letter'
12	FS	InitLex	41	–<	48
13	–<	21	42	FS	Append
14	{	Integer	43	–<	46
15	FS	InitInteger	44	T	'Letter'
16	FS	Append	45	→	47
17	T	'Digit'	46	T	'Digit'
18	–<	16	47	→	41
19	}	Integer	48	}	Tag
20	→	83	49	–<	52
21	–<	34	50	FR	IsKeyWord
22	{	Tag	51	→	54
23	FS	InitTag	52	FR	IsModeIndication
24	T	'Letter'	53	FR	SetModeIndication
25	–<	32	54	–<	56
26	FS	Append	55	T	'Blind'
27	–<	30	56	FS	SetBlindInvisible
28	T	'Letter'	57	}	BoldTag

58	→	83	76	–<	80
59	{	SpecSymbol	77	FS	SetComma
60	–<	64	78	T	','
61	FS	SetLeftBracket	79	→	82
62	T	'['	80	FS	SetColon
63	→	82	81	T	':'
64	–<	68	82	}	SpecSymbol
65	FS	SetRightBracket	83	FS	MakeLex
66	T	']'	84	–<	92
67	→	82	85	{	Comment
68	–<	72	86	FS	SetComment
69	FS	SetLeftParenthesis	87	T	'{'
70	T	'('	88	FS	Restore
71	→	82	89	T	'}'
72	–<	76	90	}	Comment
73	FS	SetRightParenthesis	91	→	84
74	T	')'	92	end	Lexeme
75	→	82			

Словарь терминалов:

1	'Digit'	4	'Letter'	7	'['	10	')'
2	'['	5	'.'	8	']'	11	','
3	'}'	6	'Blind'	9	'('	12	':'

Словарь нетерминалов:

	Acronym:	**Synonym:**
1	Lexeme	(Lexeme)

Словарь вспомогательных понятий:

	Acronym:	**Synonym:**
1	Bold Tag	(Bold Tag)
2	Comment	(Comment)
3	Integer	(Integer)
4	Spec Symbol	(Spec Symbol)
5	Tag	(Tag)

Словари контекстных символов:

Forward pass semantics:

1 Append
2 InitInteger
3 InitLex
4 InitTag
5 MakeLex
6 Restore
7 SetBlindInvisible
8 SetBlindVisible
9 SetColon
10 SetComma
11 SetComment
12 SetLeftBracket
13 SetLeftParenthesis
14 SetModeIndication
15 SetRightBracket
16 SetRightParenthesis

Forward pass resolvers:

1 IsKeyWord
2 IsModeIndication

Управляющая таблица прямого просмотра GenerLex состоит из одной лишь таблицы управляющих элементов[122] (табл.7.6), поскольку соответствующий процессор — конечный. По этой же причине в ней отсутствует графа Магазин.

Таблица 7.6

Вход	Резольвер	Семантика	Состояние
1	**2**	**3**	**4**
Состояние 1={0}, 6={10}, 14={7}			
Digit		InitLex; InitInteger; Append	2
{		SetComment	3
Letter		InitLex; InitTag	4
.		InitLex; SetBlindVisible	5
Blind			6
[		InitLex; SetLeftBracket	7
]		InitLex; SetRightBracket	8
(		InitLex; SetLeftParenthesis	9
)		InitLex; SetRightParenthesis	10
,		InitLex; SetComma	11
:		InitLex; SetColon	12
Состояния 2={17} (конечное)			
ε		MakeLex	**Stop**
Digit		Append	2
{		MakeLex; SetComment	13

[122] Совмещенной с таблицей резольверов в форме дополнительной графы Резольвер в ее составе.

Продолжение табл. 7.6

1	2	3	4
Состояния 3={5}			
}		Restore	14
Состояния 4={24}, 15={30}, 16={28} (конечные)			
ε		MakeLex	**Stop**
Digit		Append	15
{		MakeLex; SetComment	13
Letter		Append	16
Состояния 5={37}			
Letter		InitTag	17
Состояния 7={62}, 8={66}, 9={70}, 10={74}, 11={78}, 12={81} (конечные)			
ε		MakeLex	**Stop**
{		MakeLex; SetComment	13
Состояния 13={87}			
}		Restore	18
Состояния 17={40}, 19={46}, 20={44} (конечные)			
ε	IsKeyWord	SetBlindInvisible; MakeLex	**Stop**
	IsModeIndication	SetModeIndication; SetBlindInvisible; MakeLex	**Stop**
Digit		Append	19
{	IsKeyWord	SetBlindInvisible; MakeLex; SetComment	13
	IsModeIndication	SetModeIndication; SetBlindInvisible; MakeLex; SetComment	13
Letter		Append	20
Blind	IsKeyWord		21
	IsModeIndication	SetModeIndication	21
Состояние 18 ={ 89 } (конечное)			
ε			
{		SetComment	13
Состояние 21 ={ 55 } (конечное)			
ε		SetBlindInvisible; MakeLex	**Stop**
{		SetBlindInvisible; MakeLex; SetComment	13

Напомним, что ε-вход используется лишь в том случае, когда текущий входной символ не числится среди входов таблички, соответствующей данному состоянию.

Если для некоторого состояния и входного символа графа Резольвер пуста, то ее следует интерпретировать как тождественно истинный предикат. В противном случае вычисляются конъюнкции предикатов, относящихся к каждой резольверной цепочке. Если ни одна из конъюнкций не есть **true**, то имеет место контекстная ошибка. Если две или более конъюнкций (не считая пустого резольверного входа) дают **true**, то имеет место контекстная неоднозначность. В любом случае пустой резольверный вход используется в последнюю очередь.

Символ **Stop** в графе Состояние означает завершение работы процессора-сканера.

Размеры компонент управляющей таблицы GenerLex представлены в табл. 7.7. Поскольку этот процессор — конечный, то ее компоненты, представляющие магазинные цепочки и магазинные символы, имеют нулевой размер. По той же причине никакой таблицы возвратных состояний не существует.

Таблица 7.7

Компонента		Элементов	Байт
Характеристики состояний:	Descr	21	84
	Pool	21	42
Лексический вход:	Descr	21	84
	Pool	32	192
Резольверный вход		6	24
Пул управляющих элементов		29	174
Семантические цепочки:	Descr	20	80
	Pool	36	72
Резольверные цепочки:	Descr	3	12
	Pool	3	6
Магазинные цепочки:	Descr	0	0
	Pool	0	0
Магазинные символы:	Descr	0	0
	Pool	0	0
Конечные состояния		15	30
Табл. возвр. состояний:	Descr	0	0
	Pool	0	0
			794

Оптимизированная управляющая таблица процессора GenerLex получается из его первоначальной управляющей таблицы (см. табл. 7.6) путем перехода в ней к классам эквивалентных состояний и входных символов. При оптимизации были использованы следующие классы эквивалентности состояний:

1: {1 6 14} 4: {4 15 16} 7: {13} 10: {21}
2: {2} 5: {5} 8: {7 19 20}
3: {3} 6: {7 8 9 10 11 12} 9: {18}

и входных символов:

1: {'Digit'} 4: {'Letter'} 7: {'['} 10: {')'}
2: {'{'} 5: {'.'} 8: {']'} 11: {','}
3: {'}'} 6: {'Blind'} 9: {'('} 12: {':'}

Здесь классы пронумерованы, а их члены заключены в фигурные скобки.

Управляющая таблица оптимизированного сканера генераторов состоит из таблицы управляющих элементов, совмещенной с таблицей резольверов в форме дополнительной графы Резольвер в ее составе (см. табл. 7.8).

Таблица 7.8

Вход	Резольвер	Семантика	Состояние
1	**2**	**3**	**4**
Класс состояний 1={1 6 14}			
[Digit]		InitLex; InitInteger; Append	2
[{]		SetComment	3
[Letter]		InitLex; InitTag	4
[.]		InitLex; SetBlindVisible	5
[Blind]			1
[[]		InitLex; SetLeftBracket	6
[]]		InitLex; SetRightBracket	6
[(]		InitLex; SetLeftParenthesis	6
[)]		InitLex; SetRightParenthesis	6
[,]		InitLex; SetComma	6
[:]		InitLex; SetColon	6
Класс состояний 2={2} (конечных)			
ε		MakeLex	**Stop**
[Digit]		Append	2
[{]		MakeLex; SetComment	7
Класс состояний 3={3}			
[}]		Restore	1
Класс состояний 4={4 15 16} (конечных)			
ε		MakeLex	**Stop**
[Digit]		Append	4
[{]		MakeLex; SetComment	7
[Letter]		Append	4

Продолжение таблицы 7.8

1	2	3	4
Класс состояний 5={5}			
[Letter]		InitTag	8
Класс состояний 6={7 8 9 10 11 12 } (конечных)			
ε		MakeLex	**Stop**
[{]		MakeLex; SetComment	7
Класс состояний 7={13}			
[}]		Restore	9
Класс состояний 8={17 19 20 } (конечных)			
ε	IsKeyWord	SetBlindInvisible; MakeLex	**Stop**
	IsModeIndication	SetModeIndication; SetBlindInvisible; MakeLex	**Stop**
[Digit]		Append	8
[{]	IsKeyWord	SetBlindInvisible; MakeLex; SetComment	7
	IsModeIndication	SetModeIndication; SetBlindInvisible; MakeLex; SetComment	7
[Letter]		Append	8
[Blind]	IsKeyWord		10
	IsModeIndication	SetModeIndication	10
Класс состояний 9 = { 18 } (конечных)			
ε			**Stop**
[{]		SetComment	7
Класс состояний 10 = { 21 } (конечных)			
ε		SetBlindInvisible; MakeLex	**Stop**
[{]		SetBlindInvisible; MakeLex; SetComment	7

Как видим, вместо 21 состояния в исходном процессоре GenerLex оптимизированный процессор имеет всего 10 классов эквивалентных состояний — входов в управляющую таблицу.

Что касается классов эквивалентности входных символов — лексических классов (Digit, Letter, Blind), то они были предусмотрены заранее без "подсказки" со стороны технологической системы и использованы в спецификации процессора. Поэтому-то никакого полезного эффекта от оптимизации лексических входов мы не наблюдаем.

Размеры компонент оптимизированной таблицы GenerLex — см. табл. 7.9.

Таблица 7.9

Компонента		Элементов	Байт
Классы состояний:	Descr	10	40
	Pool	21	42
Лексический вход:	Descr	10	40
	Pool	32	192
Резольверный вход		6	24
Пул управляющих элементов		27	162
Семантические цепочки:	Descr	20	80
	Pool	36	72
Резольверные цепочки:	Descr	2	8
	Pool	2	4
Магазинные цепочки:	Descr	0	0
	Pool	0	0
Классы магазинных символов:	Descr	0	0
	Pool	0	0
Классы конечных состояний		6	12
Табл. возвр. состояний:	Descr	0	0
	Pool	0	0
			676

Сравнивая суммарный размер компонент этой таблицы с аналогичным значением для неоптимизированного процессора, приведенным в табл. 7.7, видим, что экономия памяти благодаря оптимизации сканера GenerLex составляет 15% — почти столько же, сколько дала оптимизация главного процессора Gener (16%).

Диагностические сообщения оптимизированного процессора GenerLex представлены в следующем перечне:

В классе состояний 1:

В конструкции Lexeme
ожидается Comment, 'Blind', Integer, Tag, BoldTag, SpecSymbol

В классе состояний 2:

В конструкции Integer после компоненты 'Digit' ожидается 'Digit'

В конструкции Lexeme после компоненты Integer ожидается Comment

В классе состояний 3:

В конструкции Comment после компоненты '{' ожидается '}'

В классе состояний 4:

В конструкции Lexeme после компоненты Tag ожидается Comment

В конструкции Tag после компоненты 'Digit' ожидается 'Letter', 'Digit'

В конструкции Tag после компоненты 'Letter' ожидается 'Letter', 'Digit'

В классе состояний 5:

В конструкции BoldTag после компоненты '.' ожидается Tag

В конструкции Lexeme
после компоненты SpecSymbol ожидается Comment

В классе состояний 7:

В конструкции Comment после компоненты '{' ожидается '}'

В классе состояний 8:

В конструкции BoldTag после компоненты Tag ожидается 'Blind'

В конструкции Lexeme после компоненты BoldTag ожидается Comment

В конструкции Tag после компоненты 'Digit' ожидается 'Letter', 'Digit'

В конструкции Tag после компоненты 'Letter' ожидается 'Letter', 'Digit'

В классе состояний 9:

В конструкции Lexeme после компоненты Comment ожидается Comment

В классе состояний 10:

В конструкции Lexeme после компоненты BoldTag ожидается Comment

***Замечание* 4.** В спецификации трансляции GenerLex не были определены синонимы для нетерминала и вспомогательных понятий. Поэтому в приведенных диагностических сообщениях использованы рабочие термины (акронимы). Разумеется, в окончательной версии этого процессора следовало бы использовать русско-язычную терминологию. Здесь этого не сделано, чтобы показать, что на ранней стадии разработки трансляции выбор окончательной терминологии для диагностических сообщений временно можно отложить "на потом".

7.5. ОДНОПРОСМОТРОВЫЙ АНАЛИЗ ГЕНЕРАТОРОВ АЛГОЛА 68

Для иллюстрации многопроцессорной[123] техники трансляции здесь мы рассмотрим ход анализа на лексическом и синтаксическом уровнях одного локального генератора Алгола 68, а именно:

.loc .struct ([5]**.int** *x*, **.proc**[ ]**.real** *y*).

Лексический анализ реализуется конечным процессором GenerLex, а синтаксический — сплайновым процессором Gener. Собственно сканированием входного текста занимается транслитератор, использующий таблицу транслитерации, построенную по описанию микролексики в спецификации трансляции GenerLex. Оба процессора однопросмотровые и контекстно-чувствительные. Тот и другой используют оптимизированные управляющие таблицы (см. табл. 7.4 и 7.8).

Протокол однопросмотрового анализа генератора : .loc .struct([5].int *x*, .proc[ ].real *y*). В представленном далее протоколе (табл. 7.10) левая колонка показывает работу основного процессора Gener, а правая — процессора

[123] Имеется в виду использование нескольких языковых процессоров.

GenerLex, используемого в качестве сканера. Номера строк протокола представляют дискретное время. После литерного представления лексемы в скобках указаны номер лексического класса и номер лексемы в этом классе (для каждого процессора эта нумерация — своя).

Таблица 7.10

Уровень синтаксического анализатора (Gener):		Уровень лексического анализатора (GenerLex):	
1		**2**	
0	**Состояние** 1		
		1	**Состояние**: 1
		2	Лексема: '.' (5,1)
		3	... принята в состоянии 1
		4	Семантики: InitLex SetBlindVisible
		5	**Состояние**: 5
		6	Лексема: 'l' (4,12)
		7	... принята в состоянии 5
		8	Семантики: InitTag
		9	**Состояние**: 8
		10	Лексема: 'o' (4,15)
		11	... принята в состоянии 8
		12	Семантики: Append
		13	**Состояние**: 8
		14	Лексема: 'c' (4,3)
		15	... принята в состоянии 8
		16	Семантики: Append
		17	**Состояние**: 8
		18	Лексема: ' ' (6,1)
		19	Резольверы: Is KeyWord (**true**)
		20	Резольверы: IsModeIndication (**false**)
		21	... принята в состоянии 8
		22	**Состояние**: 10
		23	Лексема: '.' (5,1)
		24	Семантики: SetBlindInvisible MakeLex
25	Лексема: '.loc' (1,0)		
26	... принята в состоянии 1		
27	Семантики: Init SetActual		
28	**Состояние**: 2		
		29	**Состояние**: 1
		30	Лексема: '.' (5,1)
		31	... принята в состоянии 1
		32	Семантики: InitLex SetBlindVisible
		33	**Состояние**: 5
		34	Лексема: 's' (4,19)
		35	... принята в состоянии 5
		36	Семантики: InitTag
		37	**Состояние**: 8
		38	Лексема: 't' (4,20)
		39	... принята в состоянии 8
		40	Семантики: Append
		41	**Состояние**: 8

Продолжение табл. 7.10

1		2	
		42	Лексема: 'r' (4,18)
		43	... принята в состоянии 8
		44	Семантики: Append
		45	**Состояние**: 8
		46	Лексема: 'u' (4,21)
		47	... принята в состоянии 8
		48	Семантики: Append
		49	**Состояние**: 8
		50	Лексема: 'c' (4,3)
		51	... принята в состоянии 8
		52	Семантики: Append
		53	**Состояние**: 8
		54	Лексема: 't' (4,20)
		55	... принята в состоянии 8
		56	Семантики: Append
		57	**Состояние**: 8
		58	Лексема: '(' (9,1)
		59	Резольверы : IsKeyWord(**true**)
		60	Резольверы : IsModeIndication (**false**)
		61	Семантики: SetBlindInvisible MakeLex
62	Лексема: '.struct' (5,0)		
63	... принята в состоянии 2		
64	**Состояние**: 5		
		65	**Состояние**: 1
		66	Лексема: '(' (9,1)
		67	... принята в состоянии 1
		68	Семантики: InitLex SetLeftParenthesis
		69	**Состояние**: 6
		70	Лексема: '[' (7,1)
		71	Семантики: MakeLex
72	Лексема: '(' (6,0)		
73	... принята в состоянии 5		
74	**Состояние**: 10		
		75	**Состояние**: 1
		76	Лексема: '[' (7,1)
		77	... принята в состоянии 1
		78	Семантики: InitLex SetLeftBracket
		79	**Состояние**: 6
		80	Лексема: '5' (1,6)
		81	Семантики: MakeLex
82	Лексема: '[' (11,0)		
83	... принята в состоянии 10		
84	**Состояние**: 7		
		85	**Состояние**: 1
		86	Лексема: '5' (1,6)
		87	... принята в состоянии 1

Продолжение табл. 7.10

1		2	
		88	Семантики: InitLex InitInteger Append
		89	**Состояние**: 2
		90	Лексема: ']' (8,1)
		91	Семантики: MakeLex
92	Лексема: '5' (14,0)		
93	Резольверы: IsActual (**true**)		
94	... принята в состоянии 7		
95	**Состояние**: 13		
		96	**Состояние**: 1
		97	Лексема: ']' (8,1)
		98	... принята в состоянии 1
		99	Семантики: InitLex SetRightBracket
		100	**Состояние**: 6
		101	Лексема: '.' (5,1)
		102	Семантики: MakeLex
103	Лексема: ']' (13,0)		
104	... принята в состоянии 13		
105	**Состояние**: 12		
		106	**Состояние**: 1
		107	Лексема: '.' (5,1)
		108	... принята в состоянии 1
		109	Семантики: InitLex SetBlindVisible
		110	**Состояние**: 5
		111	Лексема: 'i' (4,9)
		112	... принята в состоянии 5
		113	Семантики: InitTag
		114	**Состояние**: 8
		115	Лексема: 'n' (4,14)
		116	... принята в состоянии 8
		117	Семантики: Append
		118	**Состояние**: 8
		119	Лексема: 't' (4,20)
		120	... принята в состоянии 8
		121	Семантики: Append
		122	**Состояние**: 8
		123	Лексема: ' ' (6,1)
		124	Резольверы: IsKeyWord (**false**)
		125	Резольверы: IsModeIndication (**true**)
		126	... принята в состоянии 8
		127	Семантики: SetModeIndication
		128	**Состояние**: 10
		129	Лексема: 'x' (4,24)
		130	Семантики: SetBlindInvisible MakeLex
131	Лексема: '.int' (3,0)		
132	... принята в состоянии 12		
133	**Состояние**: 3		
134	**Состояние**: 3		
135	**Состояние**: 19		

Продолжение табл. 7.10

1		2	
		136	**Состояние**: 1
		137	Лексема: 'x' (4,24)
		138	... принята в состоянии 1
		139	Семантики: InitLex InitTag
		140	**Состояние**: 4
		141	Лексема: ',' (11,1)
		142	Семантики: MakeLex
143	Лексема: 'x' (8,0)		
144	... принята в состоянии 19		
145	**Состояние**: 22		
		146	**Состояние**: 1
		147	Лексема: ',' (11,1)
		148	... принята в состоянии 1
		149	Семантики: InitLex SetComma
		150	**Состояние**: 6
		151	Лексема: '.' (5,1)
		152	Семантики: MakeLex
153	Лексема: ',' (9,0)		
154	... принята в состоянии 22		
155	**Состояние**: 23		
		156	**Состояние**: 1
		157	Лексема: '.' (5,1)
		158	... принята в состоянии 1
		159	Семантики: InitLex SetBlindVisible
		160	**Состояние**: 5
		161	Лексема: 'p' (4,16)
		162	... принята в состоянии 5
		163	Семантики: InitTag
		164	**Состояние**: 8
		165	Лексема: 'r' (4,18)
		166	... принята в состоянии 8
		167	Семантики: Append
		168	**Состояние**: 8
		169	Лексема: 'o' (4,15)
		170	... принята в состоянии 8
		171	Семантики: Append
		172	**Состояние**: 8
		173	Лексема: 'c' (4,3)
		174	... принята в состоянии 8
		175	Семантики: Append
		176	**Состояние**: 8
		177	Лексема: '[' (7,1)
		178	Резольверы: IsKeyWord(true)
		179	Резольверы: IsModeIndication (**false**)
		180	Семантики: SetBlindInvisible MakeLex
181	Лексема: '.proc' (15,0)		
182	... принята в состоянии 23		
183	Семантики: SetFormal		
184	**Состояние**: 8		

Продолжение табл. 7.10

1		2	
		185	**Состояние**: 1
		186	Лексема: '[' (7,1)
		187	... принята в состоянии 1
		188	Семантики: InitLex SetLeftBracket
		189	**Состояние**: 6
		190	Лексема: ']' (8,1)
		191	Семантики: MakeLex
192	Лексема: '[' (11,0)		
193	... принята в состоянии 8		
194	**Состояние**: 7		
		195	**Состояние**: 1
		196	Лексема: ']' (8,1)
		197	... принята в состоянии 1
		198	Семантики: InitLex SetRightBracket
		199	**Состояние**: 6
		200	Лексема: '.' (5,1)
		201	Семантики: MakeLex
202	Лексема: ']' (13,0)		
203	Резольверы: IsNotActual (**true**)		
204	... принята в состоян 7		
205	**Состояние**: 12		
		206	**Состояние**: 1
		207	Лексема: '.' (5,1)
		208	... принята в состоянии 1
		209	Семантики: InitLex SetBlindVisible
		210	**Состояние**: 5
		211	Лексема: 'r' (4,18)
		212	... принята в состоянии 5
		213	Семантики: InitTag
		214	**Состояние**: 8
		215	Лексема: 'e' (4,5)
		216	... принята в состоянии 8
		217	Семантики: Append
		218	**Состояние**: 8
		219	Лексема: 'a' (4,1)
		220	... принята в состоянии 8
		221	Семантики: Append
		222	**Состояние**: 8
		223	Лексема: 'l' (4,12)
		224	... принята в состоянии 8
		225	Семантики: Append
		226	**Состояние**: 8
		227	Лексема: ' ' (6,1)
		228	Резольверы: IsKeyWord(**false**)
		229	Резольверы: IsModeIndication (**true**)
		230	... принята в состоянии 8
		231	Семантики: SetModeIndication
		232	**Состояние**: 10
		233	Лексема: 'y' (4,25)

Продолжение табл. 7.10

1		2	
		234	Семантики: SetBlindInvisible MakeLex
235	Лексема: '.real' (3,0)		
236	... принята в состоянии 12		
237	**Состояние**: 3		
238	**Состояние**: 3		
239	**Состояние**: 18		
240	Семантики: Reset		
241	**Состояние**: 19		
		242	**Состояние**: 1
		243	Лексема: 'y' (4,25)
		244	... принята в состоянии 1
		245	Семантики: InitLex InitTag
		246	**Состояние**: 4
		247	Лексема: ')' (10,1)
		248	Семантики: MakeLex
249	Лексема: 'y' (8,0)		
250	... принята в состоянии 19		
251	**Состояние**: 22		
		252	**Состояние**: 1
		253	Лексема: ')' (10,1)
		254	... принята в состоянии 1
		255	Семантики: InitLex SetRightParenthesis
		256	**Состояние**: 6
		257	Лексема: '**Eof**' (0,0)
		258	Семантики: MakeLex
259	Лексема: ')' (7,0)		
260	... принята в состоянии 22		
261	**Состояние**: 3		
262	**Состояние**: 17		
263	Семантики: Complete		
264	Выполнение завершено в конечном состоянии 17		

Полезно пронаблюдать, что после того, как процессор Gener принимает очередную лексему, вызывается сканер GenerLex, который, начав работу со своего начального состояния 1, заканчивает ее вызовом семантики MakeLex, которая и формирует ожидаемую основным процессором лексему. Можно также заметить, что процессор GenerLex иногда завершает свою работу, приняв входную лексему, а иногда не принимая ее — и тогда эта же входная лексема повторно анализируется при следующем вызове сканера. Разное завершение зависит от того, является ли текущая литера последней литерой формируемой лексемы или она — первая литера следующей.

Бросающиеся в глаза подряд сменяющие друг друга состояния процессора Gener без приема входной лексемы, например в моменты 133-134, 237-240, 261, соответствуют ε-движениям.

7.6. ЧЕЛНОЧНЫЙ АНАЛИЗАТОР ГЕНЕРАТОРОВ АЛГОЛА 68

Проиллюстрируем возможность использования челночных процессоров для получения синтаксической структуры входных цепочек. Очевидно, что в этом случае необходимо использовать неоптимизированный челночный процессор. В дополнение к управляющей таблице прямого просмотра (см. табл.7.1–7.2) потребуется расширенная таблица обратного, содержащая структурную информацию о вспомогательных конструкциях.

Расширенная управляющая таблица обратного просмотра процессора Gener отличается от обычной тем, что в ней присутствует графа Структурные метки (см табл. 7.11). Во время интерпретации управляющего элемента челночный процессор на обратном просмотре выдает содержимое этой графы на выход в качестве структурных скобок. Эти скобки представляют границы терминальных порождений вспомогательных понятий, входящих в состав анализируемой цепочки.

Аналогичные структурные скобки служат для ограничения терминальных порождений нетерминалов. Причем скобка, отмечающая конец конструкции выдается обратным просмотром на выход в момент, когда он помещает символ в магазин, а скобка, отмечающая ее начало — в момент выборки этого символа из магазина. Имена соответствующих нетерминалов доступны обратному просмотру через характеристики магазинных символов — возвратных состояний и алфавит нетерминалов в составе управляющей граф-схемы.

Графы Резольверы и Семантика в табл. 7.11 не показаны, так как на обратном просмотре в процессоре Gener контекстные символы не используются.

Таблица 7.11

Вход	Магазин	Структурные метки	Состояние
1	**2**	**3**	**4**
Состояние 1 = {11}			
St20	3	**конец** - ACTUAL DECLARER	2
Состояние 2 = {106}			
St4			4
St21	5	**конец** - DECLARATOR **конец** - REFERENCE TO MODE DECLARATOR **конец** - VIRTUAL DECLARER	2
St22	6	**конец** - DECLARATOR **конец** - PROCEDURE DECLARATOR **конец** - FORMAL DECLARER	2
St24	7	**конец** - DECLARATOR **конец** - ROWS OF MODE DECLARATOR	2
St30		**конец** - DECLARATOR **конец** - UNION OF MODE DECLARATOR	8
St32		**конец** - DECLARATOR **конец** - STRUCTURED WITH FIELDS DECLARATOR	9

Продолжение табл. 7.11

1	2	3	4
Состояние 3 = {8}			
St2		**начало** - ACTUAL DECLARER	10
St3		**начало** - ACTUAL DECLARER	11
Состояние 4 = {14}			
ε			**Pop**
Состояние 5 = {22}			
St5		**начало** - VIRTUAL DECLARER	12
Состояние 6 = {87}			
St9		**начало** - FORMAL DECLARER	13
St28		**начало** - FORMAL DECLARER	14
Состояние 7 = {70}			
St14		**конец** - ROWER	15
Состояние 8 = {102}			
St26	16	**конец** - FORMAL DECLARER	2
Состояние 9 = {41}			
St27		**конец** - PORTRAYER	17
Состояния 10 = {4}, 11 = {6} (конечные)			
St1		**конец** - PORTRAYER	**Stop**
Состояние 12 = {20}			
ε		**начало** - REFERENCE TO MODE DECLARATOR **начало** - DECLARATOR	**Pop**
Состояние 13 = {76}			
ε		**начало** - PROCEDURE DECLARATOR **начало** - DECLARATOR	**Pop**
Состояние 14 = {85}			
St25	18	**конец** - FORMAL DECLARER	2
Состояние 15 = {68}, 20 = {66}			
St8			19
St12			20
St13			21
St15		**конец** - UNIT	22
St19		**конец** - UNIT	22
Состояние 16 = {97}			
St17		**начало** - FORMAL DECLARER	23
St31		**начало** - FORMAL DECLARER	24

Продолжение табл. 7.11

1	2	3	4
Состояние 17 = {33}			
St23 St33	25		2 26
Состояние 18 = {80}			
St16 St29		**начало** - FORMAL DECLARER **начало** - FORMAL DECLARER	27 28
Состояние 19 = {50}			
ε St7		**начало** - ROWER **начало** - ROWS OF MODE DECLARATOR **начало** - DECLARATOR **начало** - ROWER	**Pop** 29
Состояние 21 = {64}			
St8 St12			19 20
Состояние 22 = {59}			
St8 St12 St18		**начало** - UNIT **начало** - UNIT **начало** - UNIT	19 20 30
Состояние 23 = {95}			
St10			31
Состояние 24 = {100}			
St26	16	**конец** - FORMAL DECLARER	2
Состояние 25 = {32}			
St11 St33		**начало** - PORTRAYER	32 33
Состояние 26 = {35}			
St27			17
Состояние 27 = {78}			
St9			13
Состояние 28 = {83}			
St25	18	**конец** - FORMAL DECLARER	2
Состояние 29 = {48}			
ε		**начало** - ROWS OF MODE DECLARATOR **начало** - DECLARATOR	**Pop**

Продолжение табл. 7.11

1	2	3	4
Состояние 30 = {57}			
St15		**конец** - UNIT	34
Состояние 31 = {94}			
ε		**начало** - UNION OF MODE DECLARATOR **начало** - DECLARATOR	**Pop**
Состояние 32 = {30}			
St6			35
Состояние 33 = {38}			
St27			17
Состояние 34 = {55}			
St8 St12		**начало** - UNIT **начало** - UNIT	19 20
Состояние 35 = {29}			
ε		**начало** - STRUCTURED WITH FIELDS DECLARATOR **начало** - DECLARATOR	**Pop**

Порядок интерпретации управляющего элемента расширенной таблицы обратного просмотра несколько отличается от обычного и состоит в следующем:

1. Если текущее состояние характеризуется множеством терминальных вершин — а они всегда помечены одинаковыми терминальными символами, то это множество выдается на выход в виде номеров записей граф-схемы, представляющих эти вершины, и общей терминальной метки. Любая из этих вершин может считаться породившей данный терминальный символ.

2. Если текущее состояние характеризуется множеством нетерминальных вершин, то это множество выдается на выход в виде номеров записей, представляющих эти вершины, и соответствующих нетерминальных меток — информационных полей этих записей с префиксом **начало**-. Такого рода состояния становятся текущими только в результате их выборки из магазина.

3. Если текущее состояние характеризуется множеством конечных вершин граф-схемы, то никакой информации об этом состоянии на выход не выдается.

4. Текущий входной символ (состояние прямого просмотра) анализируется в текущем состоянии обратного просмотра. Если символ принимается, то данный магазинный символ, если он задан, записывается в магазин просмотра, на выход записывается его характеристика в виде номеров нетерминальных вершин и их меток с префиксом **конец**-, исполняется семантическая цепочка, если она не пуста, на выход выдаются структурные метки вспомогательных понятий в той последовательности, в которой они заданы в управляющем элементе, и, наконец, текущее состояние принимает заданное значение. Далее производится анализ характеристики нового текущего состояния (см. шаги 1-3).

5. Если текущий входной символ не принимается, то исполняется семантическая цепочка, на выход выдаются заданные структурные метки, а новое текущее состояние извлекается из магазина. В нем повторно анализируется тот же входной символ (см. шаг 2).

Челночный анализатор Gener на том же входе, но с использованием неоптимизированных таблиц прямого (7.1) и обратного прямого (7.11) просмотров, в режиме генерации структуры входной цепочки дает результат, показанный в графе Обратный просмотр табл. 7.12.

Протокол челночного анализа локального генератора:
.loc .struct ([5] **.int** *x*, **.proc [ ] .real** *y*).

Таблица 7.12

Прямой просмотр			Обратный просмотр			
№	Сост.	Вход	№	Сост.	Вход	Выход (структурированный вход)
1	**2**	**3**	**4**	**5**	**6**	**7**
1	1	**.loc**	28	10	1	4 - ‘**.loc**’
2	2	**.struct**	27	3	2	**начало**-ACTUAL DECLARER **начало**-8-DECLARER
3	6	(	26	35		**начало**-STRUCTURED WITH FIELDS DECLARATOR 29 - ‘**.struct**’
4	11	[	25	32	6	30 - ‘(‘
5	8	5	24	25	11	**начало**-PORTRAYER **начало**-32-DECLARER
6	15	]	23	19		**начало**-DECLARATOR **начало**-ROWS OF MODE DECLARATOR **начало**-ROWER 50 - ‘[‘
7	14	**.int**	22	22	8	**начало**-UNIT 59 - ‘5’
8	4		21	15	15	**конец**-UNIT 68 - ‘]’
9	24		20	7	14	**конец**-ROWER **начало**-70-DECLARER
10	23	*x*	19	4		14 - ‘**.int**’
11	27	,	18	2	4	
12	33	**.proc**	17	2	24	**конец**-70-DECLARER **конец**-ROWS OF MODE DECLARATOR **конец**-DECLARATOR

Продолжение табл. 7.12

1	2	3	4	5	6	7
13	9	[	16	17	23	**конец**-32-DECLARER 33 - '*x*'
14	8	]	15	33	27	38 - ','
15	14	**.real**	14	25	33	**начало**-32-DECLARER
16	4		13	13		**начало**-DECLARATOR **начало**-PROCEDURE DECLARATOR 76 - '**.proc**'
17	24		12	6	9	**начало**-FORMAL DECLARER **начало**-87-DECLARER
18	22		11	19		**начало**-DECLARATOR **начало**-ROWS OF MODE DECLARATOR **начало**-ROWER 50 - '['
19	23	*y*	10	15	8	68 - ']'
20	27	)	9	7	14	**конец**-ROWER
21	32		8	4		**начало**-70-DECLARER
22	20		7	2	4	14 - '**.real**'
23			6	2	24	**конец**-ROWS OF MODE DECLARATOR **конец**-DECLARATOR **конец**-70-DECLARER
24			5	2	22	**конец**-FORMAL DECLARER **конец**-PROCEDURE DECLARATOR **конец**-DECLARATOR **конец**-87-DECLARER
25			4	17	23	**конец**-32-DECLARER 33 - '*y*'
26			3	9	27	**конец**-PORTRAYER 41 - ')'
27			2	2	32	**конец**-STRUCTURED WITH FIELDS DECLARATOR **конец**-DECLARATOR
28			1	1	20	**конец**-8-DECLARER **конец**-ACTUAL DECLARER

Протокол состоит из двух полос, одна из которых представляет работу прямого просмотра, а другая — обратного. Нумерация шагов определяет последовательность сканирования входных лексем на прямом просмотре и его состояний — на обратном. Соответственно полосу прямого просмотра следует читать сверху вниз, а обратного — снизу вверх. Результат анализа образуется на обратном просмотре (он показан в последней графе табл. 7.12). При входных

символах и граничных марках терминальных порождений Нетер-миналов указаны номера соответствующих вершин управляющей граф-схемы Gener. Они имеются в характеристиках состояний обратного просмотра одноименного процессора.

7.7. СООБЩЕНИЯ ОБ ОШИБКАХ ВО ВРЕМЯ ПРОЦЕССИРОВАНИЯ

Ошибки, обнаруживаемые при процессировании, бывают двух видов: *бесконтекстные* и *контекстные*. Механизм автоматического обнаружения и диагностики тех и других "встроен" в управляющий процессор. Но в отличие от исчерпывающе полной диагностики бесконтекстных ошибок автоматическая диагностика контекстных ошибок возможна только в форме перечисления резольверов, соответствующих невыполненным контекстным условиям или тем нескольким предикатам, выполнение которых приводит к контекстной неоднозначности. Выдача развернутых диагностических сообщений о такого рода ошибках обычно программируется в самих резольверах или семантиках.

Сообщения о бесконтекстных ошибках. Если на входе процессора любого уровня обнаруживается бесконтекстная ошибка — а обнаруживается она только в неподавляемых состояниях, то процессор выдает диагностические сообщения, относящиеся ко всем синтаксическим уровням конструкций, охватывающих место ошибки. Оно отмечается курсором в исходном тексте. Стандартный формат сообщения, относящегося к одному синтаксическому уровню, имеет вид:

В конструкции ... после компоненты ... ожидается ...

Вместо многоточий в таком сообщении используются синонимы нетерминалов и вспомогательных понятий[124] или терминальные символы (вернее, обозначения их лексических классов). При желании термины, образующие рамку, могут быть изменены. Например, можно задать рамку на английском языке или пропустить вводные слова (конструкции, компоненты). В последнем случае (на русском языке) в редакторе диагностических сообщений придется переделать окончания для согласования предлогов с последующими терминами. При таком редактировании рамка может варьироваться в каждом сообщении, так что фактически она не будет видна.

Анализ локального генератора с бесконтекстной ошибкой. В предыдущем разделе был приведен протокол анализа локального генератора **.loc .struct** ([5] **.int** *x*, **.proc** [ ] **.real** *y*) без ошибок. Сделаем в этом примере преднамеренно ошибку, пропустив стробирующую точку в ключевом слове **.proc**. Тогда proc без предшествующей точки будет восприниматься просто как возможный указатель поля структуры. Но тогда последующая скобка неуместна.

[124] Синонимы — это термины в скобках, следующие за акронимами в списках нетерминальных и вспомогательных символов в TSL-спецификации.

И в самом деле, подсистема процессирования высвечивает этот символ во входной цепочке как ошибочный (рис. 7.2.), приглашая нажать кнопку '↓' для получения развернутого диагностического сообщения.

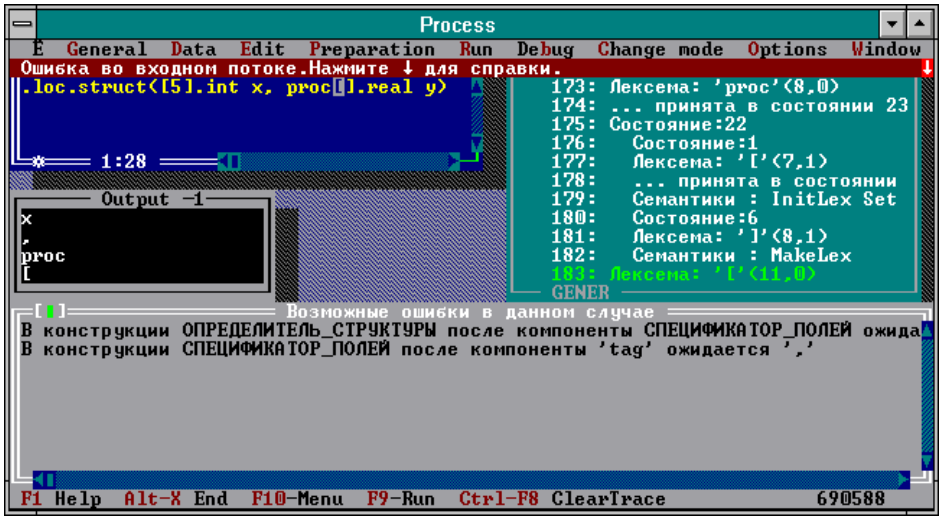


Рис. 7.2. Вид панели подсистемы процессирования
в момент обнаружения бесконтекстной ошибки.

На панели подсистемы процессирования Process видны четыре окна: окно ввода входной цепочки (вверху слева), окно трассировки или протокола (вверху справа), окно результата (Output) и окно диагностических сообщений о бесконтекстной ошибке. Во входном окне виден текст локального генератора с курсором на "ошибочном" символе ('['). В окне протокола зафиксированы последние шаги анализа перед обнаружением ошибки (ошибка была обнаружена на шаге 183). В выходном окне видны несколько последних символов перед обнаружением ошибки. Их выдача обеспечивается семантикой MakeLex (в частности, в протоколе на шагах 182-183 видно, что лексема '[' сформирована именно этой семантикой). Наконец, в нижнем окне видны два варианта сообщений об ошибке, одна из которых объясняется на уровне конструкции ОПРЕДЕЛИТЕЛЬ СТРУКТУРЫ, а другая — на уровне конструкции СПЕЦИФИКАТОР ПОЛЕЙ. Заметим, что окно диагностических сообщений об ошибках раскрывается только после нажатия клавиши ↓ по приглашению: "Ошибка во входном потоке. Нажмите ↓ для справки.", появляющемуся в красной строке в момент обнаружения ошибки.

Сообщения о контекстных ошибках. Они обнаруживаются процессором по резольверным входам в управляющую таблицу. Каждый такой вход соответствует некоторому логическому условию, которое может выполняться или не выполняться при текущем состоянии операционной среды. Один вид контекстной ошибки характеризуется тем, что *ни одно из условий не выполняется*. Другой вид ошибки — *контекстная неоднозначность*. Она имеет место тогда, когда одновременно выполняются два или более проверяемых условий. В любом из этих случаев процессор диагностирует ошибку в терминах соответствующих резольверных символов.

Анализ локального генератора с контекстной ошибкой. Для иллюстрации механизма обнаружения и диагностики контекстных ошибок рассмотрим анализ локального генератора, о котором шла речь в предыдущем примере, преднамеренно пропустив строгую верхнюю границу[125] (5), обязательную в фактическом описателе массива, определяющего вид поля *x*, т.е. обсудим анализ входной цепочки: **.loc .struct** ([ ] **.int** *x*, **.proc** [ ] **.int** *y*).

По TSL-спецификации трансляции генераторов (см. разд. 6.4) легко понять, что сорт текущего описателя (фактический, формальный или виртуальный) отслеживается во время сканирования входной цепочки с помощью магазина — одного из элементов операционной среды.

В момент сканирования первого вхождения символа ']' проверяется единственное условие IsNotActual, имеющееся на резольверном входе состояния 7 для этого входного символа, и оно оказывается не выполненным. Другими словами, в тот момент, когда уже был принят предшествующий символ '[' и сорт описателя *фактический*, на вход поступает символ ']', который не допустим в этой ситуации, ибо в фактическом описателе массива после символа '[' ожидается строгая граница (в нашей модели — целое). Эта ситуация хорошо видна в окне трассировки (Trace) на рис. 7.3 (на шаге 91 указано, что интерпретация резольверного символа IsNotActual дает **false**). Во входном окне (Input) место ошибки отмечено курсором на символе ']', о котором шла речь. В выходном окне (Output) видны несколько последних символов перед тем, на котором была обнаружена контекстная ошибка. В дополнительном окне Error сообщается тип ошибки: контекстная.

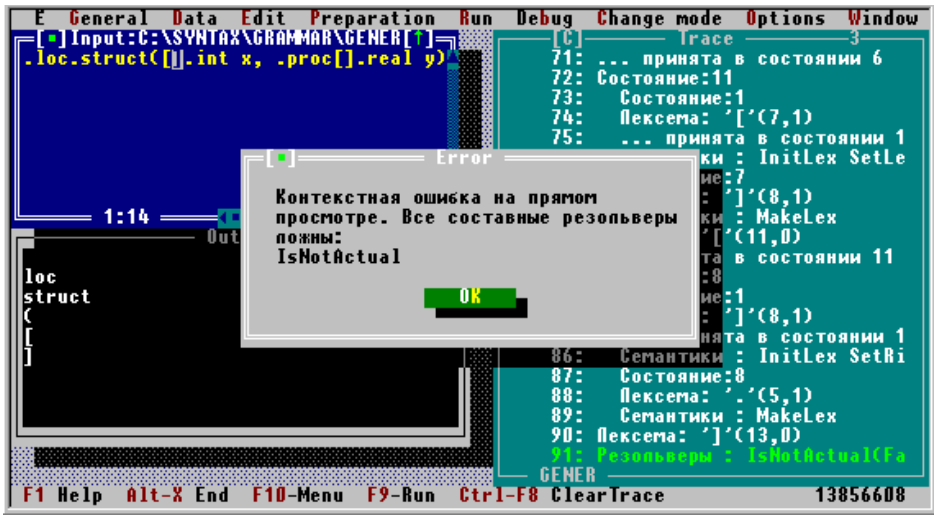


Рис. 7.3. Панель подсистемы процессирования
в момент обнаружения контекстной ошибки.

[125] Впрочем, программисты часто делают эту ошибку не преднамеренно.

Разумеется, было бы лучше предусмотреть такую реализацию резольвера IsNotActual, которая выдавала бы расшифровку контекстной ошибки в выходном окне. Выполнить соответствующую модификацию реализации этого резольвера (в спецификации операционной среды Gener) предлагается читателю самостоятельно.

Контекстная неоднозначность показана на рис. 7.4. Пример взят из языка Object Pascal — входного языка системы программирования DELPHI. Спецификация этого языка, здесь не показанная, была не полна в том отношении, что в ней еще не была определена обработка контекстных условий. В частности, резольверные символы, связанные с условиями идентификации имен, временно были ассоциированы с тождественно истинными предикатами. Поэтому в окне Input как контекстно-неоднозначное выделено имя t, хотя оно фактически и описано как имя типа. Из трех предикатов, связанных с резольверами, показанными в окне Error, должен выполняться только один — IsType, а фактически оказались выполненными все три.

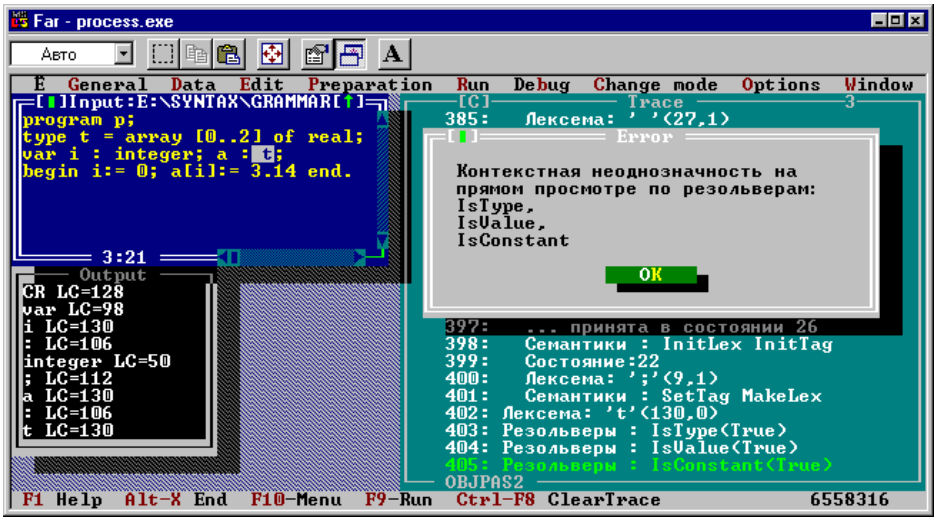


Рис. 7.4. Панель подсистемы процессирования
в момент обнаружения контекстной неоднозначности.

Эта ситуация видна также в окне трассировки (Trace): на шагах 403-405 результаты вычисления предикатов IsType, IsValue и IsConstant все — **true**.

7.8. МЕХАНИЗМ СООБЩЕНИЙ — СРЕДСТВО ОБМЕНА ИНФОРМАЦИЕЙ МЕЖДУ ПРОЦЕССОРАМИ

Из предыдущего изложения явствует, что при многопроцессорной трансляции каждый языковой процессор работает в своей локальной операционной среде. Единственный канал обмена информацией между ними, который обсуждался до сих пор, это лексические буферы, доступные процессорам соседних уровней. Но лексические буферы предназначены для передачи информации

фиксированного формата, а именно лексем от процессора-сканера процессору-анализатору.

Очевидно, что в сложных случаях желательно, чтобы один процессор имел возможность доступа к информации из локального окружения другого процессора для ее использования или модификации. Решение этой проблемы в технологическом комплексе SYNTAX основано на заимствовании концепции сообщений из MS WINDOWS.

Любой процессор может послать сообщение процессору низшего уровня (своему сканеру), процессору высшего уровня (анализатору), для которого сам он служит сканером, или самому себе. Посылка сообщения возможна из любой точки операционной среды процессора (семантики, резольвера, вспомогательной процедуры или функции) посредством вызова встроенной в систему процессирования функции Message[126]:

```
Function Message (Receiver   : TAddr;
                  Command : word;
                  WParam   : word;
                  LParam    : LongInt ) : LongInt;
```

Ее параметры: Receiver — получатель сообщения; Command — идентификатор сообщения (команда)[127] WParam и LParam — параметры сообщения (команды). Тип Taddr описывается как перечислимый скалярный тип:

```
type Taddr = ( Parser , Self, Scanner );
```

Если Receiver = Parser, то получатель сообщения — *процессор-анализатор*. Если Receiver = Self, то получатель сообщения — *сам процессор, посылающий сообщение*. Если Receiver = Scanner, то получатель сообщения — *процессор-сканер*. По значению параметра Command определяется обработчик данного сообщения (интерпретатор команды). Параметры WParam и LParam поставляют собственно информацию обработчику данного сообщения ("параметризуют" команду Command).

Функция Message формирует сообщение — запись типа TProcMessage, в соответствующие поля которой "укладывает" значения своих параметров. Этот тип определяется следующим образом:

```
type TProcMessage = record
                      Sender, Receiver : TAddr;
                      Message : word;
                      case integer of
                      0 : ( Wparam : word;
                            Lparam   : Longint;
                            Result    : LongInt);
                      1: ( WParamLo, WParamHi : byte;
                            LParamLo, LParamHi   : word;
                            ResultLo, ResultHi       : word)
                     end
```

[126] Она автоматически включается в текст библиотеки динамической загрузки.

[127] Целое от 1 до 50000. Нумерация сообщений локальна для каждого процессора.

Затем функция Message обращается к встроенному диспетчеру сообщений, который по полю Receiver производит поиск адресата и передает управление его дешифровщику команд. Тот, в свою очередь, производит вызов соответствующего обработчика сообщения (интерпретатора команды Command), передавая ему запись-сообщение как параметр. Используя значения полей WParam и LParam, обработчик в своем окружении вырабатывает результат и размещает его в поле Result той же самой записи-сообщения[128]. Наконец, функция Message возвращает результат, извлекая его из поля Result. Он используется, если необходимо, в той процедуре или функции, которая вызывала функцию Message.

Надо иметь в виду, что если обработчик сообщения не был найден[129], то функция Message возвращает 0. В противном случае функция Message извлекает свой результат из поля Result. Очевидно, что этот результат будет не определенен, если обработчик не присвоит этому полю никакого значения.

Типы полей параметров и результата записи-сообщения позволяют процессорам непосредственно обмениваться информацией объемом лишь в 4 байта. При необходимости передачи большего объема информации обработчик данного сообщения и процедура, вызывающая функцию Message, несмотря на объявленную типизацию этих полей, могут согласованно трактовать поля WParam и LParam как указатель на область данных в операционной среде процессора-отправителя сообщения, а поле Result — как указатель на область данных в операционной среде процессора-получателя сообщения.

Естественно, что обработчики сообщений находятся в операционной среде процессора, которому эти сообщения адресуются. Их описание включается в соответствующую спецификацию трансляции в специальный раздел **MESSAGES**, следующий за разделом **IMPLEMENTATION**.

В разделе **MESSAGES** для каждого сообщения, адресуемого процессору, реализующему трансляцию, которая определяется данной спецификацией, описывается свой обработчик по следующей форме:

```
Message: id_mes
          …
begin … end;
```

где *id_mes* — идентификатор принимаемого сообщения (именно он указывается в параметре Command функции Message). Для удобства программирования таким идентификатором может служить мнемоническая константа, определяемая в разделе **ENVIRONMENT**. Идентификационные номера обработчиков сообщений (id_mes) для одного процессора должны быть различны. Символ ";" обязателен даже после последнего описания обработчика сообщений в разделе **MESSAGES**.

В каждую процедуру реакции на сообщение передается неявный параметр Msg : TProcMessage. Именно он и является той записью-сообщением, которая

[128] Для удобства программирования обработчика сообщений предусмотрена альтернативная структура полей — параметров и результата.

[129] В случае, например, когда пользователь забыл его определить.

используется в качестве носителя информации от процессора-отправителя к процессору-получателю и обратно. Соответственно в исходный текст библиотеки операционной среды включается описание вида

```
procedure <RANDOM_NAME> (var Msg : TProcMessage); export;
begin … end;
```

а в предложение **exports** добавляется строка вида:

```
exports
<RANDOM_NAME> index id_First + id_Mes ;
```

где <RANDOM_NAME> — локально уникальное (в операционной среде одного процессора) имя, генерируемое автоматически подсистемой процессирования во время формирования текста библиотеки по спецификации операционной среды процессора[130]. Константа id_First равна 10000. Добавление этой константы необходимо для выхода из диапазона индексов, зарезервированных для экспортируемых из библиотеки процедур внутрисистемного назначения.

Процессоры, обменивающиеся сообщениями. Рассмотрим в качестве примера комплекс процессоров, анализирующих уже знакомые нам генераторы Алгола 68, усложнив их синтаксис. Именно, в позиции границ фактических описателей видов наряду с целыми числами (вопреки синтаксису реального языка) разрешим писать тексты форматов, разумеется, в очень упрощенном модельном представлении. Напомним, что форматы как значения особого вида в конечном итоге используются в стандартных процедурах форматного обмена.

Возникающая здесь проблема состоит в том, что некоторые символы в текстах форматов лексически должны трактоваться не так, как во всех других конструкциях языка. Например, некоторые буквы, такие как l, n, p. x, y, z, на лексическом уровне во всех конструкциях, кроме текстов форматов, должны восприниматься просто как буквы[131], а в текстах форматов — как представ-ления особых символов управления форматированием[132] значений, участвующих в процедурах обмена. Трудность состоит в том, что сканер должен однозначно относить каждое вхождение любой из этих букв к соответствующему лексическому классу, которые по определению не могут пересекаться. Очевидно, без использования контекстной информации, доступной анализатору, на уровне сканера с этой задачей не справиться. Именно здесь и может помочь механизм обмена сообщениями между процессорами разных уровней.

Действительно, анализатору в момент получения символа $, с которого начинается текст формата, достаточно послать сообщение сканеру о том, что он должен теперь воспринимать упомянутые буквы как символы размещения, а в момент получения другого вхождения символа $, которым заканчивается текст формата, послать сообщение сканеру о необходимости обычной обработки всех букв.

---

[130] При выполнении команды Process/Preparation/Compile текст этой библиотеки компилируется в исполняемый DLL-модуль. В подсистеме процессирования он используется с расширением der, а не dll.

[131] Сканер должен относить их к лексическому классу ‘буква’.

[132] И, соответственно, сканер должен формировать лексемы, относящиеся к лексическому классу ‘символ размещения’.

Организация такого рода взаимодействия между анализатором и сканером подробно обсуждается после приводимых далее полных текстов расширенных спецификаций трансляций, реализуемых процессором-анализатором (XGen) и процессором-сканером (XGenLex) соответственно.

Расширенный синтаксис генератора XGen:

LEXICS: XGenLex

SYNTAX

Nonterminals: GENERATOR (ГЕНЕРАТОР),
DECLARER (ОПИСАТЕЛЬ),
UNIT (ОСНОВА).

Auxiliary notions:

DECLARATOR (ОПРЕДЕЛИТЕЛЬ),
REFRENCE TO MODE DECLARATOR (ОПРЕДЕЛИТЕЛЬ_ИМЕНИ),
STRUCTURED WITH FIELDS DECLARATOR (ОПРЕДЕЛИТЕЛЬ_СТРУКТУРЫ),
ROWS OF MODE DECLARATOR (ОПРЕДЕЛИТЕЛЬ_МАССИВА),
PROCEDURE DECLARATOR (ОПРЕДЕЛИТЕЛЬ_ПРОЦЕДУРЫ),
UNION OF MODE DECLARATOR (ОПРЕДЕЛИТЕЛЬ_ОБЪЕДИНЕНИЯ),
PORTRAYER (СПЕЦИФИКАТОР_ПОЛЕЙ),
ROWER (ИНДЕКСАТОР),
FORMAT (ФОРМАТ),
ELEMENT SEQUENCE (ПОСЛЕДОВАТЕЛЬНОСТЬ_ЭЛЕМЕНТОВ),
FORMAT ELEMENT (ЭЛЕМЕНТ_ФОРМАТА),
REPLICATOR (ПОВТОРИТЕЛЬ).

Forward pass semantics:

Init, SetActual, SetFormal, SetVirtual, Reset, Complete,
SetSpecScanModeOn, SetSpecScanModeOff.

Forward pass resolvers : IsVIRACT, IsActual, IsNotActual.

GENERATOR : Init, Set Actual, ('.loc' ; '.heap'), DECLARER, Complete.

DECLARER : 'ModeIndication'; DECLARATOR.

DECLARATOR : REFRENCE TO MODE DECLARATOR;
STRUCTURED WITH FIELDS DECLARATOR;
ROWS OF MODE DECLARATOR;
PROCEDURE DECLARATOR;
UNION OF MODE DECLARATOR.

REFRENCE TO MODE DECLARATOR : SetVirtual, '.ref', DECLARER, Reset.

STRUCTURED WITH FIELDS DECLARATOR : '.struct', '(',PORTRAYER, ')'.

PORTRAYER : (DECLARER , 'tag' # ',') # ','.

ROWS OF MODE DECLARATOR : [IsVIRACT, '.flex'], ROWER, DECLARER.

ROWER : '[', (IsActual, [UNIT, ':'], UNIT; IsNotActual , [':']) # ',', ']'.

```
UNIT : 'integer' ; FORMAT.
PROCEDURE DECLARATOR : Set Formal , '.proc',
                                  ['(', DECLARER # ',' , ')'], DECLARER,   Reset.
UNION OF MODE DECLARATOR : SetFormal, '.union',
                                      '(', DECLARER # ',' , ')', Reset.
FORMAT : SetSpecScanModeOn,
          '$', ELEMENT SEQUENCE, SetSpecScanModeOff, '$'.
ELEMENT SEQUENCE: ( [ REPLICATOR ], FORMAT ELEMENT )+.
FORMAT ELEMENT: 'tag'.
REPLICATOR: '(', UNIT, ')'.
```

ENVIRONMENT

```
const n = 10;
      ms_SetMode = 1; sm_SpecMode = 1; sm_RealMode = 2;
type  TSort = (Actual, Formal, Virtual); TLevel = 0..n; TIndex = 1..n;
      TStack = array [1..n] of TSORT;
var Stack : TStack; t : TLevel;
```

IMPLEMENTATION

```
            { Резольверы }
function IsVIRACT : Boolean;
begin IsVIRACT := (Stack [t] = Virtual) or (Stack [t] = Actual) end;
function IsActual  : Boolean;
begin IsActual := Stack [t] = Actual end;
function IsNotActual : Boolean;
begin IsNotActual := not IsActual end;
            { Семантики }
procedure SetSpecScanModeOn;
begin Message (Scanner, ms_SetMode, sm_SpecMode, 0) end;
procedure SetSpecScanModeOff;
begin Message (Scanner, ms_SetMode, sm_RealMode, 0) end;
procedure Init;
begin t := 0 end;
procedure Reset;
begin Dec (t) end;
procedure SetActual;
begin  Inc (t); Stack[t] := Actual end;
procedure SetFormal;
begin  Inc (t); Stack[t] := Formal end;
procedure SetVirtual;
begin Inc (t); Stack[t] := Virtual end;
procedure Complete;
begin  NewLine;  PrintString ('Анализ генератора завершен успешно!') end;
```

Расширенная лексика генератора XGenLex:

MICROLEXICS

Lexical classes: '[', '{', '}', ']', '(', ')', ':', ',', '.', Letter, Digit, Blind,
Escaped Symbols, '$'.
Letter : 'a'..'z'.
Digit : '0'..'9'.
Blind: #32, #9.
Escaped Symbols: #13, #10.

SYNTAX

Nonterminals: Lexeme.

Auxiliary notions: Integer, Tag, BoldTag, SpecSymbol,
Comment, Formatter symbol.

Forward pass semantics:
SetBlindVisible, SetBlindInvisible, SetModeIndication,
SetLeftBracket, SetRightBracket,
SetLeftParanthesis, SetRightParanthesis,
SetComma, SetColon,
InitLex, MakeLex, InitInteger, InitTag, Append,
SetComment, Restore,
SetFormSymbol, SetFormator.

Forward pass resolvers : IsKeyWord, IsModeIndication,
IsFormScanMode,
IsNotFormScanMode.

Lexeme: Comment* , InitLex,
(Integer ; Tag ; BoldTag ; SpecSymbol ; FormatterSymbol) , MakeLex ,
Comment* .

Integer : InitInteger, (Append, 'Digit')$^{+}$.

Comment : SetComment , '{' , Restore , '}'.

Tag : InitTag, (IsNotFormScanMode ,'Letter' ,
(Append, ('Letter' ; 'Digit'))*) ;
(IsFormScanMode , SetFormator, 'Letter').

BoldTag : SetBlindVisible, '.', Tag,
(IsKeyWord ; IsModeIndication, SetModeIndication),
['Blind'], SetBlindInvisible.

SpecSymbol : SetLeftBracket, '[' ; SetRightBracket, ']' ;
SetLeftParenthesis, '(' ; SetRightParenthesis, ')' ;
SetComma, ',' ; SetColon, ':'.

Formatter symbol : SetFormSymbol , '$'.

ENVIRONMENT

```
const
   BlindItem  = 6; RightBrace = 3; KeyWordNmb = 7;
   ms_SetMode = 1;
   FormScan : Boolean = false;
   KeyWord : array[1..KeyWordNmb] of string = ('loc', 'heap', 'ref', 'flex', 'struct',
                                               'proc', union');
var Repr : String; LexClass : Integer;
```

IMPLEMENTATION

```
function IsKeyWord : Boolean;
var i : 0..KeyWordNmb; AllScanned, Found : Boolean;
begin  AllScanned := False;  Found := False; i := 0;
 repeat
 Inc(i); AllScanned := i = KeyWordNmb; Found := KeyWord [i] = Repr;
 if Found then begin
                  case i of
                   1 {loc}     : LexClass :=  1 ;
                   2 {heap}  : LexClass :=  2 ;
                   3 {ref}     : LexClass :=  4 ;
                   4 {flex}    : LexClass := 10 ;
                   5 {struct} : LexClass :=  5 ;
                   6 {proc}   : LexClass := 15 ;
                   7 {union} : LexClass := 16 ;
                  end
                 end
 until Found or AllScanned;
 IsKeyWord := Found
end;
function IsModeIndication : Boolean;
begin IsModeIndication := not IsKeyWord end;
function IsFormScanMode : Boolean;
begin IsFormScanMode := FormScan end;
function IsNotFormScanMode : Boolean;
begin IsNotFormScanMode := not IsFormScanMode end;
procedure SetBlindVisible;   begin SetVisibleOn (BlindItem) end;
procedure SetBlindInvisible; begin SetVisibleOff (BlindItem) end;
procedure SetModeIndication;
begin  LexClass := 3; Repr := '.' + Repr end;
procedure SetComment;
begin SetInvisible; SetVisibleOn(RightBrace) end;
procedure SetFormator;
begin NewLine;
 if LR = 'z'
 then begin LexClass := 8; Repr := 'z' end
 else PrintString('НЕИЗВЕСТНЫЙ СИМВОЛ ФОРМАТА'); NewLine
end;
```

```
procedure Restore;
begin SetVisible;  SetVisibleOff (BlindItem) end;
procedure Append;
begin Repr := Repr + LR end;
procedure InitLex;
begin Repr := '';  LexClass := –1 end;
procedure InitInteger;
begin LexClass := 14; Repr := '' end;
procedure MakeLex;
begin
  OutLR := Repr; OutLN := 0; OutLC := LexClass;
  NewLine;  PrintString (OutLR)
end;
procedure SetColon;
begin LexClass := 12; Repr := ':' end;
procedure SetComma;
begin LexClass := 9; Repr := ',' end;
procedure SetLeftBracket;
begin LexClass := 11; Repr := '[' end;
procedure SetLeftParanthesis;
begin LexClass := 6; Repr := '(' end;
procedure SetRightBracket;
begin LexClass := 13;  Repr := ']' end;
procedure SetFormSymbol;
begin LexClass := 17;  Repr := '$' end;
procedure SetRightParanthesis;
begin LexClass := 7; Repr := ')' end;
procedure InitTag;
begin Repr := LR; LexClass := 8 end;
```

MESSAGES

```
Message : ms_SetMode
begin FormScan := Msg.wParam = 1 end;
```

Опустим подробности проектирования трансляций и заострим внимание на реализации механизма межпроцессорного взаимодействия анализатора и сканера.

Как видно из правила для конструкции FORMAT в приведенной выше TSL-спецификации XGen, анализатор, принимая левый ограничитель текста формата $, выполняет семантику SetSpecScanModeOn, которая путем вызова

```
Message (Scanner, ms_SetMode, sm_SpecMode, 0)
```

сообщает сканеру, что он должен перейти в специальный режим обработки элементов формата.

Обработчик этого сообщения ms_SetMode получает через поле WParam значение sm_SpecMode = 1, и проанализировав его, устанавливает в среде сканера флажок FormScan в состояние **true**, тем самым фиксируя режим работы сканера "в тексте формата".

Аналогично, принимая на входе правый ограничитель формата $, анализатор выполняет семантику SetSpecScanModeOff, которая посредством вызова

```
Message ( Scanner, ms_SetMode, sm_RealMode, 0)
```

тому же обработчику ms_SetMode сообщает через поле WParam значение sm_RealMode = 2. Проанализировав это поле сообщения, обработчик ms_SetMode устанавливает флажок FormScan в состояние **false**, переводя тем самым сканер в режим "вне текста формата". От состояния этого флажка зависит способ обработки сканером конструкции Tag. Правило, определяющее эту конструкцию в TSL-спецификации XGenLex, имеет две альтернативы, выбираемые по резольверам IsNotFormScanMode и IsFormScanMode, которые тестируют текущее состояние флажка FormScan, установленное обработчиком сообщений ms_SetMode.

Заметим, что оба вызова функции Message передают обработчику сообщений ms_SetMode через запись-сообщение фактически только один параметр (в поле WParam), тогда как не используемое в данном случае поле LParam имеет значение 0. Значение результата функции Message вызывающими ее семантиками не используется. Поэтому обработчик сообщений ms_SetMode, принадлежащий сканеру XGenLex, поле Result в записи-сообщении оставляет неопределенным.

Глава 8

ОПИСАНИЕ ЯЗЫКА TSL

8.1. ОБЩАЯ СТРУКТУРА TSL-СПЕЦИФИКАЦИИ

A **T**ranslation **S**pecification **L**anguage (TSL) является метаязыком для записи спецификаций трансляций и одновременно служит входным языком технологического комплекса SYNTAX. Структура TSL-спецификации трансляции может быть описана при помощи следующего правила грамматики:

```
SPECIFICATION:
    [HEADER],
    [MICROLEXICS DECLARATION; LEXICS DECLARATION],
    SYNTAX DECLARATION,
    [ENVIRONMENT DECLARATION].
```

Другими словами, TSL-*спецификация некоторой трансляции состоит из необязательного заголовка, необязательного описания микролексики или лексики, обязательного описания синтаксиса и необязательного описания операционной среды*.

Заголовок (HEADER) является разновидностью комментария и может использоваться, например, в качестве заглавия или аннотации специфицируемой трансляции.

Описание микролексики (MICROLEXICS DECLARATION) определяет классификацию литер, используемых для представления входных цепочек. Ее одновременно можно рассматривать и как спецификацию транслитератора, поскольку по ней генерируются микролексические классы, используемые транслитератором, встроенным в подсистему процессирования, для формирования микролексем.

Описание лексики (LEXICS DECLARATION) фактически представляется именем другой TSL-спецификации.

Описание синтаксиса (SYNTAX DECLARATION), задает обработку конструкций, распознаваемых процессором-анализатором. По существу это описание есть некоторая управляющая RBNF-грамматика. Роль терминалов в ней играют микролексические классы, определенные непосредственно в разделе описания микролексики данной спецификации или косвенно в указанной спецификации лексики. По описанию синтаксиса подсистема проектирования строит управляющие таблицы процессора, реализующего трансляцию, задаваемую данной спецификацией.

Описание операционной среды (ENVIRONMENT DECLARATION) определяет семантику входного языка в терминах процедур и функций, ассоциированных с контекстными символами управляющей грамматики, и данных, над которыми они оперируют. Это описание является диалектом некоторого языка программирования[133].

8.2. ЗАГОЛОВОК

Заголовок (HEADER) является общим комментарием к спецификации трансляции. В заголовке можно указать, например, имя грамматики, информацию о том, какую трансляцию она определяет, номер версии грамматики, дату выпуска и т.п. Заголовок фактически не имеет фиксированной синтаксической структуры и системой не анализируется.

Синтаксис

HEADER : 'Любая цепочка литер, предшествующая ключевому слову в начале следующего раздела описаний'.

Пример:

```
CALC — Грамматика калькулятора
Версия : 23.02.1994
```

8.3. КОММЕНТАРИЙ

Комментарий — это информация, адресованная человеку, которая игнорируется системой SYNTAX. Комментарий можно помещать почти в любом месте спецификации. Однако не допускаются комментарии внутри терминалов и синонимов. "Вложенные" комментарии, т.е. комментарии внутри комментариев, также не допустимы.

Синтаксис

```
COMMENT: '{' , 'произвольная последовательность символов' , '}' ;
         '%', 'произвольная последовательность символов' , '↵'.
```
[134]

Другими словами, комментарий — это любая последовательность символов, не содержащая в себе символов "{" и "}", заключенная в фигурные скобки "{", "}" (она может занимать несколько строчек текста), или любая цепочка символов, ограниченная слева символом процента "%", а справа символом конца строки (невидимым

Примеры:

```
Formula : L {Левый операнд}, 'Op' {Операция}, R {Правый операнд} .

% П Р А В И Л А
```

[133] В настоящее время в качестве такого базового языка программирования используется Паскаль, реализуемый компилятором **bpc** фирмы Борланд.

[134] Терминал '↵' обозначает конец строки.

8.4. ЛЕКСИКА ЯЗЫКА TSL

Буквы используются в идентификаторах и терминалах.

Синтаксис

LETTER : 'a'; 'b'; 'c'; 'd'; 'e'; 'f '; 'g'; 'h'; 'i'; 'j'; 'k'; 'l'; 'm';
'n'; 'o'; 'p'; 'q'; 'r'; 's'; 't'; 'u'; 'v'; 'w'; 'x'; 'y'; 'z';
'A'; 'B'; 'C'; 'D'; 'E'; 'F'; 'G'; 'H'; 'I'; 'J'; 'K'; 'L'; 'M';
'N'; 'O'; 'P'; 'Q'; 'R'; 'S'; 'T'; 'U'; 'V'; 'W'; 'X'; 'Y'; 'Z' ;
'а'; 'б'; 'в'; 'г'; 'д'; 'е'; 'ж'; 'з'; 'и'; 'к'; 'л'; 'м'; 'н'; 'о'; 'п'; 'р';
'с'; 'т'; 'у'; 'ф'; 'х'; 'ц'; 'ч'; 'ш'; 'щ'; 'ы'; 'ъ'; 'ь'; 'э'; 'ю'; 'я';
'А'; 'Б'; 'В'; 'Г'; 'Д'; 'Ж'; 'З'; 'И'; 'Й'; 'К'; 'Л'; 'М'; 'Н'; 'О'; 'П'; 'Р';
'С'; 'Т'; 'У'; 'Ф'; 'Х'; 'Ц'; 'Ч'; 'Ш'; 'Щ'; 'Ъ'; 'Ы'; 'Ь'; 'Э'; 'Ю'; 'Я';
'@' ; '$' ; '_' {подчеркивание}.

Другими словами, буквами считаются прописные и строчные буквы латинского и русского алфавитов, а также символы '@', '$' и '_' (подчеркивание)[135].

Цифры используются в составе элементов словарей грамматики, т.е. нетерминалов, терминалов и контекстных символов.

Синтаксис

DIGIT : '0' ; '1' ; '2' ; '3' ; '4 ; '5' ; '6' ; '7' ; '8' ; '9' .

Специальные знаки используются в правилах грамматики в качестве метасинтаксических знаков или для обозначения регулярных операций.

Синтаксис

Special symbol : '(' ; ')' ; ':' ; ',' ; '.' ; '+' ; '*' ; '#' ; ';' ; '&' ; '[' ; ']' .

В *круглых скобках* записываются синоним нетерминала или вспомогательного понятия; операнд регулярного выражения в правой части правила, если необходимо задать явным образом порядок выполнения регулярных операций, независимо от их старшинства.

Двоеточие отделяет ключевое слово от списка элементов в описаниях алфавитов грамматики; левую часть правила от правой.

Запятая разделяет элементы алфавитов; используется в качестве операции конкатенации в регулярном выражении.

Точка завершает описания алфавитов или правила грамматики.

Плюс Клини (+) обозначает регулярную операцию усеченной итерации.

Звездочка Клини (*) обозначает регулярную операцию итерации или замыкания.

Звездочка Цейтина (#) обозначает регулярную операцию итерации с разделителем.

Точка с запятой обозначает регулярную операцию объединения.

Амперсанд (&) обозначает пустой операнд, порождающий пустую цепочку.

---

[135] Подчеркивание рекомендуется использовать в синонимах вместо пробела, ибо пробелы исключаются из диагностических терминов.

В *квадратные скобки* заключается "необязательная" часть правила. Подразумевается, что член правила, заключенный в такие скобки, помимо порождения, определяемого регулярным выражением внутри скобок, порождает еще и пустую цепочку. Как и круглые скобки, они влияют на порядок “вычисления” регулярного выражения.

Пример:

```
Real Number : [IP] {Необязательная целая часть} ,
              FP  {Дробная часть}.
```

Ключевые слова служат для разметки разделов TSL-спецификации и идентификации списков символов, составляющих словари трансляционной грамматики. Ключевое слово можно представлять прописными или строчными буквами. Оно может содержать произвольное число пробелов (например, его можно написать вразрядку). Естественно, что свободно выбираемые символы не должны совпадать с ключевыми словами.

Синтаксис[136]

MICROLEXICS	{Начало раздела описания микролексики}
LEXICS	{Начало раздела описания лексики }
Lexical classes	{Начало списка микролексических классов}
SYNTAX	{Начало раздела описания синтаксиса}
Terminals	{Начало списка терминалов}
Nonterminals	{Начало списка нетерминалов}
Auxiliary notions	{Начало списка вспомогательных понятий}
Forward pass semantics	{Начало списка семантик прямого просмотра}
Backward pass semantics	{Начало списка семантик обратного просмотра}
Forward pass resolvers	{Начало списка резольверов прямого просмотра}
Backward pass resolvers	{Начало списка резольверов обратного просмотра}
ENVIRONMENT	{Начало раздела описания операционной среды}
IMPLEMENTATION	{Начало раздела интерпретации контекстных символов}

Идентификаторы служат для именования микролексических классов, а также обозначения нетерминалов, вспомогательных понятий[137], семантик и резольверов.

Синтаксис

TAG : LETTER , (LETTER ; DIGIT)*.

Другими словами, идентификатор — это последовательность букв и цифр, начинающаяся с буквы. Прописные и строчные буквы различаются. Пробелы могут выставляться между любыми символами идентификатора (например, их можно писать вразрядку), не влияя на их идентификацию. Идентификатор должен располагаться в одной строчке спецификации.

---

[136] Ключевые слова перечисляются в порядке их использования в TSL-спецификации.

[137] Точнее, они используются в качестве акронимов и синонимов.

Примеры:

```
Expression     Integral  Part
F21            IntegralPart
```

Оба написания Integral Part (с пробелом) и IntegralPart (без пробела) воспринимаются как один и тот же идентификатор.

Терминалы определяются своим появлением в словаре терминальных символов или в правых частях правил грамматики. В спецификации трансляции они скорее играют роль лексических классов[138], чем символов, непосредственно составляющих предложение языка.

Синтаксис

```
TERMINAL : " ' " , DENOTATION  , " ' " ;  ' " ' ,  DENOTATION  ,  ' " ' .
DENOTATION : 'Любая цепочка литер'.
```

Другими словами, терминал — это произвольная последовательность литер, заключенная между двумя одинаковыми ограничителями, в качестве которых можно использовать апострофы (') или двойные кавычки ("). Терминал должен заканчиваться на той же строчке, на которой начинается. Два разных вида ограничителей дают возможность использовать эти символы в составе терминалов.

Примеры:

```
':='          " ' " {Апостоф с двойных кавычках}                'd'
'begin'       ' " ' {Двойные кавычки между апостофами}          "2"
```

Синонимы используются в качестве грамматических терминов в диагностических сообщениях об ошибках на входе процессора. Они могут определяться для нетерминалов и вспомогательных понятий грамматики и позволяют формулировать сообщения об ошибках с их привязкой к синтаксической структуре входного языка в терминах, понятных пользователям. Синонимы входят в описания соответствующих словарей, а в правилах грамматики не используются. Следует иметь в виду, что пробелы, встречающиеся в синонимах, игнорируются. Рекомендуется использовать символ подчеркивания '_' в качестве видимого пробела.

Синтаксис

```
SYNONYM : TAG.
```

Пример:

```
AUXILIARY NOTIONS :      IP ( целая _часть ) ,
                         FP ( дробная _часть ) .
```

Здесь целая_часть и дробная_часть — синонимы вспомогательных понятий (IP и FP — акронимы).

---

[138] Именно они используются транслитератором или сканером при формировании входной лексемы для анализатора.

8.5. ОПИСАНИЕ МИКРОЛЕКСИКИ

Транслитерация. Описание микролексики специфицирует *транслитератор* — языковой процессор самого нижнего уровня, который каждую литеру из входного потока относит к одному из микролексических классов. Транслитератор преобразует входную литеру в микролексему, т. е. структуру из двух целочисленных кодов, представляющих номер микролексического класса и номер элемента этого класса; третье поле этой структуры — сама входная литера. Описание микролексики как раз и определяет упомянутые микролексические классы.

Описание микролексики необходимо при спецификации *сканеров* — языковых процессоров, распознающих конкретные представления терминальных символов грамматики (основных символов входного языка), или простейших (регулярных) конструкций языка, таких, как литеральные константы и идентификаторы.

Синтаксис

MICROLEXICS DECLARATION : '**MICROLEXICS**',
LEXICAL CLASSES, LEXICAL CLASS DECLARATION*.

LEXICAL CLASSES : '**Lexical classes**' , ':' , ITEM LIST, '.' .

LEXICAL CLASS DECLARATION : TAG , ':' , [(LITERAL ; SUBRANGE) # ','] , '.' .

ITEM LIST : ITEM # ',' .

ITEM : TAG ; LITERAL ; SUBRANGE .

TAG : LETTER, (LETTER ; DIGIT)* .

SUBRANGE : LOWER BOUND, '..' , UPPER BOUND.

LOWER BOUND : LITERAL.

UPPER BOUND : LITERAL.

LITERAL : ' " ' , (TOKEN ; " ' ") , ' " ' ; " ' " , (TOKEN ; ' " ') ," ' " ; '#', ASCII CODE .

TOKEN : Любая представимая литера .

ASCII CODE : Целое из диапазона 0..255 .

Раздел описания микролексики начинается символом **MICROLEXICS**. После этого символа следует список микролексических классов (LEXICAL CLASSES), т.е. терминальных символов, использующихся в управляющей грамматике[139], предваряемый ключевой фразой **Lexical classes** и двоеточием. Элементы списка разделяются запятыми. Список заканчивается точкой. Порядок перечисления классов не существенен, поскольку их нумерация определяется порядком появления соответствующих им терминальных символов в описании синтаксиса в этой же спецификации трансляции.

[139] С точки зрения управляющей грамматики (микро) лексический класс является терминальным символом. Способ обработки всех входных (микро) лексем одного класса — один и тот же.

Различаются *литераные* и *именованные* микролексические классы. В списке микролексических классов они могут всречаться в любом порядке.

Литеральный класс представляется литерой, заключенной в апострофы, и не нуждается в дальнейшем описании. Для описания литеры ' (апостроф) используется форма "'" (апостроф между двумя двойными кавычками). Для описания литеры " (двойные кавычки) используется форма '"' (двойные кавычки между двумя апострофами). Допускается использовать сокращенное описание нескольких литеральных классов, если ASCII-коды их элементов-литер имеют последовательные значения. Такое описание состоит из начального и конечного литералов, разделенных специальным знаком ".." (две точки подряд). Например, следующее описание:

Lexical classes : 'A' , '0'..'9'.

определяет набор из 11 (одноэлементных) литеральных классов.

Именованные классы представляются в списке микролексических классов их именами (идентификаторами без апострофов и кавычек). Имя класса, заключенное в апострофы или двойные кавычки, используется в правилах управляющей грамматики в качестве терминала[140] наряду с литеральными микролексическими классами.

Каждое имя именованного микролексического класса должно определяться соответствующим описанием (LEXICAL CLASS DECLARATION). Эти описания следуют за списком микролексических классов.

Состав именованного класса определяется правилом. Указывается имя класса (из списка классов), затем следует двоеточие и список элементов класса, который записывается по тем же правилам, что и список литеральных классов в предложении **Lexical classes**. Но для спецификации литер, не отображаемых на экране, служит другая форма представления — это знак '#', после которого следует десятичное число из диапазона 0..255, обозначающее ASCII-код литеры. Порядковый номер[141] элемента в списке элементов класса транслитератором упаковывается во второе поле микролексемы (первое поле — номер микролексического класса).

Существует специальный именованный класс с фиксированным именем Escaped symbols, который определяет множество символов, игнорируемых сканером (если это необходимо); в нем также зафиксирован специальный символ " (два апострофа подряд), обозначающий конец входного потока литер (аналог Eof).

Описания именованных классов, включая Escaped symbols, должны следовать за списком микролексических классов. Микролексические классы не должны пересекаться. Все перечисленные в предложении **Lexical classes** именованные классы и только они должны быть определены (в любом порядке). Никакой класс, кроме специального класса Escaped symbols, не может быть пустым.

---

[140] Это можно видеть в примере из разд. 1.3.

[141] Нумерация элементов класса начинается с единицы.

Пример:

```
MICROLEXICS
Lexical classes : 'A' , 'B' , 'C' , 'a' .. 'c' , blank , '[' ,  ' " ' , " ' " , digit,
                  Escaped symbols ,  Eof.
blank :  #32.
digit :  '0' .. '9'.
Eof :  " {Два подряд идущий апострофа}.
Escaped symbols : #10 , #13 , '*' .
```

В приведенном примере описана микролексика, состоящая из 13 классов, из которых четыре (blank, digit, Eof и Escaped symbols) — именованные, а остальные ('A', 'B', 'C', 'a', 'b', 'c' , '[', ' " ' и " ' ") — литеральные. Класс blank имеет один элемент (пробел), класс ‘digit’ имеет 10 элементов, класс Eof имеет один элемент, а класс Escaped symbols — три элемента, причем первые два есть коды перевода строки и возврата каретки. Поскольку элементы классов нумеруются от 1, то, например, '2' в классе ‘digit’ имеет номер 3.

8.6. ОПИСАНИЕ ЛЕКСИКИ

Раздел описания лексики используется в том случае, если для реализации специфицируемой трансляции в качестве сканера используется процессор, а не транслитератор. Тогда после заголовка раздела описания лексики следует указать имя спецификации трансляции, реализуемой этим сканером.

Синтаксис

```
LEXICS DECLARATION : 'LEXICS' , ':', SPECIFICATION NAME.
SPECIFICATION NAME : TAG.
```

Пример:

```
LEXICS : GenLex
```

Такое описание лексики использовано в спецификации генераторов Алгола 68 (см. разд. 6.4). Под именем GenLex числится спецификация сканера для анализатора генераторов.

8.7. ОПИСАНИЕ СИНТАКСИСА

Раздел описания синтаксиса является источником информации для построения управляющей таблицы конечного или сплайнового процессора, реализующего специфицируемую трансляцию.

Синтаксис

```
SYNTAX DECLARATION : ['SYNTAX' ;  'PRODUCTIVE SYNTAX'],
                     NONTERMINALS LIST,
                     [AUXILIARY NOTIONS LIST],
                     [TERMINALS LIST],
                     [FORWARD PASS SEMANTICS LIST],
                     [BACKWARD PASS SEMANTICS LIST],
                     [FORWARD PASS RESOLVERS LIST],
                     [BACKWARD PASS RESOLVERS LIST],
                     RULES .
```

Описание синтаксиса входного языка специфицируемой трансляции представляется в виде RBNF-грамматики, которая состоит из алфавитов и правил. Этот раздел должен начинаться с ключевого слова **SYNTAX** — в случае анализирующей грамматики, или **PRODUCTIVE SYNTAX** — в случае, когда грамматика порождающая. Далее следуют алфавиты грамматики:

алфавит нетерминалов,
алфавит вспомогательных понятий,
алфавит терминалов,
алфавит семантик прямого просмотра,
алфавит резольверов прямого просмотра,
алфавит семантик обратного просмотра,
алфавит резольверов обратного просмотра.

Обязательным является только алфавит нетерминалов, и он должен быть первым. Другие алфавиты, если они существуют, могут следовать в произвольном порядке. Алфавиты грамматики попарно не пересекаются. Это значит, что любой символ (акроним) может находиться не более, чем в одном из указанных списков. За алфавитами следуют правила грамматики.

Алфавит нетерминалов. Задание алфавита нетерминалов обязательно, так как в любой грамматике имеется хотя бы один нетерминал. Первый элемент в списке нетерминалов считается *начальным нетерминалом* грамматики. Этот алфавит служит для идентификации нетерминалов, используемых в правилах грамматики.

Синтаксис

NONTERMINALS LIST : '**Nonterminals**' , ':' , NOTION # ',' , '.'.

NOTION : ACRONYM , [SYNONYM] .

ACRONYM : TAG .

SYNONYM : '(' , TERM , ')' .

TERM : 'Любая цепочка литер, не содержащая пробелов'.

Описание алфавита нетерминалов начинается с ключевого слова **Nonterminals**, за которым следует двоеточие и элементы списка нетерминалов, разделенные запятыми. Список элементов завершается точкой.

Каждый элемент списка нетерминалов — понятие — состоит из *акронима* и необязательного *синонима*.

Акронимы — *сокращенные обозначения нетерминалов* — используются в правилах грамматики, тогда как синонимы — *развернутые понятия грамматики* — включаются в глоссарий диагностических сообщений. Поскольку при внесении синонимов в глоссарий диагностик пробелы отбрасываются, рекомендуется вместо пробелов использовать символ подчеркивания '_'. Если синоним не указывается, то в глоссарий в качестве синонима заносится акроним.

Пример:

Nonterminals : P (ПРОГРАММА) , UI (Целое_без_знака) , DIGIT.

Здесь обозначения P, UI и DIGIT — акронимы, а ПРОГРАММА и Целое_без_знака — синонимы. Нетерминал DIGIT синонима не имеет.

Алфавит вспомогательных понятий служит для идентификации вспомогательных понятий, используемых в правилах грамматики. Описание алфавита вспомогательных понятий вполне аналогично описанию алфавита нетерминалов. Но существенная особенность трактовки вспомогательных понятий состоит в том, что их вхождения в правые части правил заменяются на определяющие их регулярные выражения во время преобразования грамматики в эквивалентную граф-схему, которая в технологическом комплексе SYNTAX является формой внутреннего представления грамматики. И хотя при такой подстановке вспомогательные понятия исключаются из правил грамматики, в граф-схеме сохраняется информация о первоначальной синтаксической структуре предложений входного языка.

Типичный прием балансировки грамматик состоит в переносе символов из алфавита нетерминалов в алфавит вспомогательных понятий. Как отмечалось, это равнозначно подстановке определений вместо используемых вхождений вспомогательных понятий[142].

Синтаксис

AUXILIARY NOTIONS LIST : '**Auxiliary notions**' , ':' , NOTION # ',' , '.'.

Другими словами, описание алфавита вспомогательных понятий начинается с ключевой фразы **Auxiliary notions**, за которой следует двоеточие и список понятий. Понятия разделяются запятыми. В конце ставится точка.

Каждое понятие состоит из акронима и необязательного синонима. В правилах грамматики используются акронимы, тогда как синонимы служат в качестве диагностических терминов. Отсутствующий синоним считается равным акрониму.

Пример:

Auxiliary notions : I (ЦЕЛОЕ) , E (ПОРЯДОК) ,
ЧИСЛО БЕЗ ЗНАКА .

Здесь I, E, ЧИСЛО БЕЗ ЗНАКА — акронимы, ЦЕЛОЕ и ПОРЯДОК — синонимы.

Алфавит терминалов. Список терминалов используется для задания определенного порядка их нумерации. Если терминалы расположить в порядке убывания частоты их встречаемости в языке, который специфицируется, то это может ускорить трансляцию этого языка.

Задавать список терминалов необязательно. В том случае, когда он отсутствует, нумерация, о которой шла речь, определяется текстуальным порядком вхождения терминалов в правые части правил грамматики. Если в данной спецификации имеется раздел описания микролексики, то соответствующий транслитератор автоматически настраивается на нумерацию терминалов, необходимую для анализатора, по нумерации, определяемой (явно или неявно) в разделе описания синтаксиса.

---

[142] Разумеется, циклические подстановки невозможны — вспомогательные понятия не могут определяться рекурсивно.

Синтаксис

TERMINALS LIST : '**Terminals**' , ':' , TERMINAL # ',' , '.' .

TERMINAL : " ' " , DENOTATION, " ' " ; ' " ' , DENOTATION, ' " ' .

DENOTATION : 'Любая цепочка литер' .

Описание алфавита терминалов начинается с ключевого слова **Terminals**, за которым следует двоеточие и элементы списка терминалов, разделенные запятыми. Список элементов завершается точкой. Элемент списка — произвольная цепочка литер, заключенная в апострофы или двойные кавычки.

Пример:

Terminals : 'begin' , ':=' , """.

Последний элемент в этом списке — апостроф в двойных кавычках.

Алфавиты семантических символов. Семантические символы ассоциируются с некоторыми преобразованиями операционной среды, выполняемыми на прямом или обратном просмотре челночного процессора. Соответственно, если семантики используются, необходимо определять алфавиты семантических символов прямого и (или) обратного просмотров.

Синтаксис

FORWARD PASS SEMANTICS LIST: '**Forward pass semantics**' , ':' , TAG # ',' , '.' .

BACKWARD PASS SEMANTICS LIST: '**Backward pass semantics**', ':', TAG # ',' , '.' .

Описания алфавитов семантик прямого и обратного просмотров различаются только ключевыми фразами, с которых они начинаются: **Forward pass semantics** или **Backward pass semantics** соответственно. Далее следует двоеточие, элементы алфавита, разделенные запятой, и точка.

Элементы-идентификаторы используются в правых частях правил грамматики в качестве семантик. В разделе описания операционной среды каждый семантический символ должен быть описан как процедура без параметров.

Примеры:

Forward pass semantics : FPInit , FPStep , FPDone .

Backward pass semantics : BPInit , BPStep , BPDone .

По этим словарям символы FPInit, FPStep, FPDone, используемые в правилах грамматики, идентифицируются как семантики прямого просмотра, тогда как символы BPInit, BPStep, BPDone — как семантики обратного.

Алфавиты резольверных символов. Резольверные символы ассоциируются с некоторыми предикатами, тестирующими текущее состояние операционной среды на прямом или обратном просмотре челночного процессора. Соответственно, если резольверы используются, необходимо определять алфавиты резольверных символов прямого и (или) обратного просмотров.

Синтаксис

FORWARD PASS RESOLVERS LIST: '**Forward pass resolvers**', ':', TAG # ',' , '.' .

BACKWARD PASS RESOLVERS LIST: '**Backward pass resolvers**', ':', TAG # ',' , '.' .

Описания алфавитов резольверов прямого и обратного просмотров различаются только ключевыми фразами, с которых они начинаются: **Forward pass resolvers** или **Backward pass resolvers** соответственно. Далее следует двоеточие, элементы алфавита, разделенные запятой, и точка.

Элементы-идентификаторы используются в правых частях правил грамматики в качестве резольверов. В разделе описания операционной среды каждый резольверный символ должен быть описан как булевская функция без параметров.

Примеры:

Forward pass resolvers : Small , Large , Ok .

Backward pass resolvers : Short , Long .

Символы Small , Large и Ok, используемые в правилах управляющей грамматики, идентифицируются как резольверы прямого просмотра, а Short , Long — как резольверы обратного.

Правила управляющей грамматики описывают синтаксическую структуру входного языка и способ его обработки. Если контекстные символы не используются, грамматика описывает бесконтекстный синтаксис входного языка. Если используются семантические символы, грамматика задает управляющую структуру, включающую обрабатывающие процедуры. Если резольверы используются, грамматика описывает также и контекстные условия. По крайней мере одно правило — правило для начального нетерминала — обязательно.

Синтаксис

```
RULES : RULE+ .
RULE : LEFT PART, ':' , RIGHT PART , '.'.
LEFT PART : NONTERMINAL ; AUXILIARY NOTION.
RIGHT PART : REGULAR EXPRESSION.
REGULAR EXPRESSION : OPERAND # DYADIC OPERATION .
OPERAND : ( NONTERMINAL;
           AUXILIARY NOTION;
           TERMINAL;
           SEMANTICS;
           RESOLVER;
           EMPTY ;
           ' (' , REGULAR EXPRESSION , ')' ;
           ' [' , REGULAR EXPRESSION , ']') ,
           MONADIC OPERATION* .
NONTERMINAL : ACRONYM { нетерминала}.
AUXILIARY NOTION : ACRONYM { впомогательного понятия}.
SEMANTICS : TAG {Семантика прямого или обратного просмотра}.
RESOLVER  : TAG { Резольвер прямого или обратного просмотра }.
EMPTY : '&' .
DYADIC OPERATION :   ';'  {Объединение} ;
                     ','  {Конкатенация} ;
                     '#'  {Итерация с разделителем}.
MONADIC OPERATION :  '*'{Итерация} ;
                     '+'  {Усеченная итерация} .
```

Правило состоит из левой и правой частей, разделенных двоеточием. В левой части может находиться нетерминал или вспомогательное понятие, а в правой — некоторое регулярное выражение над символами алфавитов, определенных в данной грамматике[143]. Нетерминалы и вспомогательные понятия, определяемые или используемые правилами, представляются своими акронимами.

Регулярные операции в двух последних правилах перечислены по возрастанию старшинства, причем одноместные операции итерации и усеченной итерации имеют одинаковое и наибольшее старшинство.

8.8. ОПИСАНИЕ ОПЕРАЦИОННОЙ СРЕДЫ

Описание семантики входного языка в спецификации трансляции представляется разделом описания операционной среды. Этот раздел содержит описание констант, типов, переменных, процедур и функций на языке Borland Pascal 7.0. Реализация этих описаний составляет операционную среду, в которой интерпретируются контекстные символы.

В операционную среду могут входить также обработчики сообщений (интерпретаторы команд). Они определяются в разделе сообщений и служат средством прямого обмена информацией между процессорами, входящими в одно и то же SYNTAX-приложение (об использовании сообщений см. разд. 7.7).

Все эти описания включаются в библиотечный модуль, который компилируется штатным компилятором **bpc**. Сгенерированная DLL-библиотека динамически присоединяется к стандартному управляющему процессору, образуя сплайновый процессор, реализующий специфицируемую трансляцию.

Синтаксис

ENVIRONMENT DECLARATION :
'**ENVIRONMENT**', ENVIRONMENT ELEMENTS,
'**IMPLEMENTATION**', CONTEXT SYMBOL INTERPRETATION,
['**MESSAGES**', COMMAND INTERPRETATION].

ENVIRONMENT ELEMENTS : {Описания констант, типов, переменных, процедур и функций, используемых в реализации семантик (действий, ассоциируемых с терминалами в порождающих грамматиках) и резольверов, на базовом языке программирования}.

CONTEXT SYMBOL INTERPRETATION : {Описание процедур, реализующих семантики (и действия, ассоциируемые с терминалами в порождающих грамматиках), а также функций, реализующих резольверы, на базовом языке программирования} .

COMMAND INTERPRETATION : COMMAND INTERPRETER^{+}.

COMMAND INTERPRETER : '**Message**', ':', COMMAND IDENTIFIER, BODY.

COMMAND IDENTIFIER : TAG.

BODY : { Некоторая реализация обработчика команды
на базовом языке программирования }.

[143] Точнее, используются терминалы, акронимы нетерминалов и вспомогательных понятий и контекстные символы.

Описание семантики, если оно имеется, состоит из раздела описания элементов операционной среды, раздела описания интерпретации контекстных символов и необязательного раздела сообщений.

Раздел описания элементов операционной среды начинается с ключевого слова **ENVIRONMENT**, за которым следует блок описаний констант, типов, переменных, функций и процедур в полном соответствии с синтаксисом блока языка Borland Pascal 7.0.

Раздел описания интерпретации контекстных символов начинается с ключевого слова **IMPLEMENTATION**, за которым следуют описания булевых функций без параметров, интерпретирующих резольверные символы, и описания процедур (также без параметров) интерпретации семантических символов. Каждый контекстный символ должен иметь соответствующую интерпретацию. В телах этих функций и процедур можно использовать элементы операционной среды, описанные в разделе **ENVIRONMENT**, а также процедуры и функции, встроенные в подсистему процессирования.

Раздел сообщений имеется в описании операционной среды каждого процессора, который получает сообщения от других процессоров, входящих в состав одного и того же SYNTAX-приложения. Он начинается с ключевого слова **MESSAGES**, за которым следует последовательность описаний обработчиков сообщений.

Описание обработчика сообщения начинается ключевым словом **Message**, за которым следует двоеточие, затем идентификатор сообщения, и, наконец, тело обработчика, завершающееся точкой с запятой. Тело обработчика — то же самое, что и тело процедуры в языке Borland Pascal 7.0. Пример использования сообщений приведен в спецификации XGenLex (см. разд. 7.8)

Многочисленные примеры описания операционной среды на Паскале, используемом в качестве базового языка программирования в технологическом комплексе SYNTAX, можно найти в предыдущих главах книги.

Глава 9

ТЕСТИРОВАНИЕ ЯЗЫКОВЫХ ПРОЦЕССОРОВ

9.1. ТЕСТИРОВАНИЕ — СПОСОБ ПРОВЕРКИ ПРАВИЛЬНОСТИ СПЕЦИФИКАЦИЙ

В современных условиях, когда практически нет такой сферы человеческой деятельности, где бы не использовались ЭВМ, надежность программного обеспечения становится важнейшим требованием. Ошибки программирования делаются все более дорогостоящими как по своим, иногда катастрофическим, последствиям, так и по стоимости их обнаружения и устранения. Анализ стоимости разработки, реализации и сопровождения больших программных систем показывает [17], что около 97% затрат на весь жизненный цикл системы приходится на производственную и операционную фазы, причем 50% общих расходов делается после того, как она достигает статуса "первого выпуска". Из этих 97% около половины затрачивается на генерацию программ и их модификаций, тогда как другая половина — на то, чтобы убедиться в их правильности.

Таким образом, современная технология производства больших программных систем не обеспечивает необходимого уровня надежности, вследствие чего они доводятся до кондиции уже в процессе эксплуатации методом "молотка и клещей". Причина этого в том, что сама технология производства программ недостаточно надежна.

Обычно под технологией подразумевается некоторый способ производства продукта определенного вида. Технология надежна, если конечный продукт, произведенный при строгом ее соблюдении, гарантированно соответствует всем предъявляемым к нему требованиям. Формулировки этих требований называют *спецификациями.*

Одним из путей повышения надежности технологии является ее обоснование, т.е. доказательство, что она обеспечивает достаточные условия для того, чтобы конечный продукт соответствовал данным спецификациям. Однако практически никогда не удается ни точно и полно сформулировать эти условия, ни доказать их достаточность. Именно неуверенность в надежности технологии, а вернее, уверенность в ее ненадежности, заставляет прибегать к услугам ОТК — вынужденной надстройке над собственно технологией, задачей которой является проверка, соответствует ли de facto продукт, произведенный при реально сложившихся условиях, данным спецификациям.

Чем позже обнаруживается дефект в выпускаемом изделии, тем дороже оно обходится изготовителю и, разумеется, потребителю, не говоря уже о прямом

вреде, который может причинить дефектное изделие. Поэтому, как правило, контролируется каждый этап производства и даже отдельные операции. Чем выше надежность собственно технологии, тем меньше роль контролирующей надстройки.

Все вышесказанное полностью относится и к производству программного продукта. Производство больших программ обходится весьма дорого и, как уже отмечалось, основные затраты идут на доводку написанных программ до работающего состояния.

Универсальной технологии программирования при наличии общих технологических принципов не существует. Каждый вид программного продукта требует своей специфической технологии производства. То, что производство программ все еще обходится слишком дорого, свидетельствует лишь о его кустарном характере.

Разработка надежной технологии программирования — трудная задача. Представляется целесообразным рассмотреть эту проблему в применении к некоторым конкретным видам программ, в первую очередь к таким, которые предназначаются для широкого и интенсивного использования и которые сами являются продуктами серийного производства. Одним из таких видов являются синтаксически управляемые программы, такие, как трансляторы языков программирования.

Сложившаяся к настоящему времени модель процесса трансляции годится для реализации многих языков программирования. Особенности конкретных языков учитываются путем параметризации этой модели.

В аспекте разработки надежной технологии реализации языков программирования с нетривиальным синтаксисом имеет смысл подробно рассмотреть весь процесс трансляции по классической схеме:

- специфицировать каждую фазу процесса трансляции;
- построить математическую модель реализации каждой фазы;
- обосновать эти модели или, по крайней мере, определить их слабые места, где нужно позаботиться о контролирующей надстройке — тестировании;
- разработать методы такого тестирования.

9.2. ТЕСТИРОВАНИЕ SYNTAX-ПРИЛОЖЕНИЙ

В предыдущих главах подробно было показано использование контекстно чувствительных конечных и сплайновых процессоров в качестве основных моделей реализации синтаксически управляемых трансляций. Можно доказать, что описанный способ их построения дает процессоры, адекватные тем спецификациям трансляции, по которым они строятся. Но нет гарантии, что спецификации правильно отражают реальные потребности конечного пользователя разрабатываемого средства синтаксически управляемой обработки данных.

Очевидно, что не существует другого пути выяснить, удовлетворяет ли заказчика конечный продукт, как только продемонстрировать его в работе на некотором конечном (но "представительном") наборе тестовых вариантов.

Облегчающим обстоятельством в этой ситуации является *многоуровневость* или расслоенность SYNTAX-технологии. В самом деле, любое SYNTAX-приложение представляет собой некоторую композицию конечных и сплайновых процессоров, на нижнем уровне которой используется типовой транслитератор. Поэтому тестирование такого приложения распадается на тестирование его компонент (транслитераторов, конечных и сплайновых процессоров) и тестирование межкомпонентных связей.

Взаимодействие транслитератора и сканера. Настройка встроенного транслитератора на реализацию желаемой микролексики производится посредством генерации микролексических классов, специфицируемых в разделе описания микролексики. Основным источником ошибок спецификации на этом уровне могло бы быть рассогласование имен микролексических классов, используемых в спецификации микролексики, и обозначений терминалов управляющей грамматики, описывающей трансляцию, реализуемую сканером. Однако проверка согласованного определения микролексики в разделе MICROLEXICS и ее использования в разделе SYNTAX, встроенная в технологический комплекс, исключает такую неприятность. Поэтому на данном технологическом участке тестирование не требуется.

Взаимодействие сканера и анализатора. Подобная же опасность существует при взаимодействии анализатора со сканером, в роли которого используется не транслитератор, а другой процессор. Здесь контроль за согласованным определением лексики и ее использованием со стороны технологического комплекса SYNTAX не столь жесткий, поскольку формирование лексем выполняется семантиками процессора-сканера. При программировании этих семантик необходимо следить за тем, чтобы были согласованы номера лексических классов, формируемые этими семантиками, и номера соответствующих лексических входов в управляющую таблицу процессора-анализатора. Внешние имена терминалов в грамматике, специфицирующей сканер, и в грамматике, специифицирующей анализатор, могут быть совершенно различными, лишь бы их нумерация была согласована, хотя такая разноголосица при проектировании увеличивает риск появления ошибки. При программировании семантик сканера рекомендуется заглядывать в словарь лексики анализатора, доступный по команде Preparation / ShowLexics в подсистеме процессирования, которая выдает занумерованный список лексических классов, в терминах которых написаны правила грамматики анализатора. Именно эти номера и должны включаться в поля LC выходных лексем сканера. Очевидно, что на этом участке SYNTAX-технологии *тестирование было бы полезно*. Оно может быть организовано путем систематической генерации тестовых вариантов по грамматике сканера (конечного процессора). Очевидно, что эта генерация воспроизводит полностью словарь основных символов языка, дополнив его "представительными" образцами литералов, идентификаторов и т.п. лексических единиц.

Постановка задачи тестирования конечных процессоров. Как отмечалось во Введении, тестирование является одним из средств проверки "слабых" мест в программном изделии. Какие места в изделии могут оказаться слабыми, в конечном счете зависит от способа его производства. Под таким углом зрения здесь мы проанализируем слабые места конечных процессоров, производимых посредством SYNTAX-технологии, и наметим соответствующую процедуру их целенаправленного тестирования.

Напомним коротко процесс изготовления конечного процессора. Сначала пишется некоторая КС-грамматика, определяющая входной язык трансляции (процессора, ее реализующего). Затем в нее внедряется некоторое количество контекстных символов, с которыми ассоциируются преобразования, определяемые в форме процедур, оперирующих над данными (семантики), и функций, тестирующих соотношения между ними (резольверы). Эти данные включаются в качестве новых элементов операционной среды. Далее производятся эквивалентные преобразования полученной таким образом трансляционной грамматики с целью ее регуляризации. При этом может быть частично изменена система контекстных символов, их расстановка в правилах и интерпретация. И, наконец, полученная спецификация трансляции в форме контекстно чувствительной RBNF-грамматики подается на вход технологического комплекса.

Из рассмотрения этой последовательности подготовки спецификации трансляции становятся очевидными следующие типы ошибок:

1) исходная КС-грамматика не порождает требуемый входной язык трансляции;

2) интерпретация контекстных символов неверна;

3) неверна расстановка контекстных символов в правилах грамматики;

4) преобразования грамматики фактически оказались не эквивалентными (т.е. полученная грамматика определяет другую трансляцию).

Ошибки типа 1 могли бы обнаруживаться уже при простом рассмотрении набора тестовых вариантов как цепочки, не соответствующие представлению заказчика о входном языке желательной для него трансляции.

Ошибки типа 2 лучше всего обнаруживать при автономном тестировании каждой процедуры или функции, ассоциированной с соответствующим контекстным символом.

Ошибки типа 3 и 4 могут обнаруживаться тестами, инициирующими достаточно интенсивное взаимодействие между семантическими процедурами. Последние, как показывает практика использования SYNTAX-технологии, обычно настолько элементарны, что их обоснование (верификацией или тестированием) не представляет большой проблемы, и зачастую их правильность самоочевидна. Основная же трудность состоит *в проверке правильности их взаимодействия при преобразованиях операционной среды*. Именно на это предлагается направить тестирование.

Исходя из этой установки, мы естественным образом приходим к следующему критерию выбора тестовых вариантов, выражающему заданную степень взаимодействия между семантиками.

Пусть задано некоторое натуральное n. Тест (т.е. некоторое конечное множество тестовых вариантов) должен активизировать как минимум однократное исполнение каждой возможной m-ки семантик, где $1 \le m \le n$, при минимальной суммарной длине составляющих его тестовых вариантов. Это значит, что при $n = 1$ тестом должна быть испытана каждая семантика по крайней мере по одному разу (разумеется в допустимой трансляционной грамматикой последовательности вызовов). При $n = 2$, кроме того, по крайней мере по разу должно быть проверено взаимодействие между компонентами каждой возможной (по спецификации) пары семантик и т.д. Натуральное n, выступающее в роли параметра сформулированного критерия, естественно принять в качестве *меры полноты теста.* Минимальный по длине тест, удовлетворяющий этому критерию, при заданном значении n, назовем *оптимальным тестом уровня или порядка* n.

Итак, наша задача состоит в том, чтобы найти способ генерации оптимального теста уровня n по заданной явнорегулярной трансляционной грамматике, специфицирующей регулярную трансляцию.

9.3. ТЕСТИРОВАНИЕ КОНЕЧНЫХ ПРОЦЕССОРОВ

Механизм автоматической генерации теста, удовлетворяющего критерию при $n = 1$, технологичнее всего построить, базируясь на использовании управляющей граф-схемы, как адекватной формы спецификации трансляции.

И действительно, в случае явнорегулярной спецификации трансляции управляющая граф-схема состоит из единственной компоненты связности — ориентированного графа[144]. В нем имеются начальная вершина, в которую не входит ни одна дуга, конечная вершина, из которой ни одна дуга не выходит, внутренние вершины, помеченные терминалами (вернее, именами лексических классов), и дуги, помеченные цепочками контекстных символов. В дальнейшем всюду, где ради краткости будут употребляться термины *граф* или *орграф*, следует подразумевать ориентированный помеченный псевдомультиграф.

Для начала будем считать, что резольверы не используются, так что контекст[145] напрямую не влияет на механизм конечного управления процессора, реализующего соответствующую трансляцию. В этом случае задача генерации теста первого порядка равносильна нахождению в данном орграфе множества маршрутов от начальной к конечной его вершине, покрывающих все его дуги, и имеющих минимальную суммарную длину.

В математическом программировании проблема нахождения таких маршрутов в графе известна как задача о китайском почтальоне, и в математических терминах она формулируется как задача нахождения в ориентированной сети максимального потока минимальной стоимости. Ее решение по существу

---

[144] Который, строго говоря, было бы уместнее назвать *ориентированным псевдомультиграфом*, поскольку в общем случае он может иметь петли и параллельные дуги.

[145] Генерация контекстно зависимых тестов в SYNTAX-технологии не разработана.

дает минимальное дополнение исходного орграфа до эйлерова, в котором затем остается только найти эйлеров цикл. Кратные вхождения начальной и конечной вершин графа в эйлеров цикл квантуют его на участки, следы от вершин которых представляют тестовые варианты, составляющие оптимальный тест заданной силы. В то время как след вершин одного такого маршрута представляет тестовый вариант[146] след составляющих его вершин и дуг образует одну управляющую цепочку, которая показывает, какая последовательность семантик активизируется данным тестовым вариантом.

Для получения множества тестовых вариантов для $n>1$ достаточно решить аналогичную задачу на конвертированном графе (аналогичном реберному или итерированному реберному), построенном по данному графу (компоненте управляющей граф-схемы).

***Определение* 1.** Пусть $\mathcal{G}$ — некоторый орграф, подобный одной компоненте связности управляющей граф-схемы. Построим по нему новый орграф такого же типа, используя алгоритм, состоящий из следующих шагов:

1. Построение реберного графа для $\mathcal{G}$. Каждую дугу a исходного графа $\mathcal{G}$ представим как вершину a' реберного графа $\mathcal{G}'$, и из вершины a' в вершину b' графа $\mathcal{G}'$ проведем дугу, если за дугой a следует дуга b в графе $\mathcal{G}$. Очевидно, что $\mathcal{G}'$ — тоже граф, возможно, с несколькими компонентами. Каждую изолированную вершину, если таковая имеется, будем считать одновременно начальной и конечной вершиной такой вырожденной компоненты.

2. Построение новой начальной вершины в $\mathcal{G}'$. Построим новую начальную вершину в $\mathcal{G}'$ и проведем из нее дуги в каждую начальную вершину $\mathcal{G}'$.

3. Построение новой конечной вершины в $\mathcal{G}'$. Построим новую конечную вершину в $\mathcal{G}'$ и проведем в нее дуги из каждой конечной вершины $\mathcal{G}'$.

Полученный таким образом граф $\mathcal{G}'$ назовем *конвертированным*. Тот факт, что он является результатом описанного уже преобразования исходного графа $\mathcal{G}'$, будем записывать следующим образом: $\mathcal{G}' = \mathrm{conv}(\mathcal{G})$.

***Определение* 2.** Положим по определению $\mathrm{conv}^0(\mathcal{G}) = \mathcal{G}$, и для любого $n>0$ определим $\mathrm{conv}^n(\mathcal{G}) = \mathrm{conv}(\mathrm{conv}^{n-1}(\mathcal{G}))$. Таким образом определенную операцию conv^n назовем *n-кратным конвертированием.*

***Определение* 3.** Пусть $\mathcal{G}=(Q, X)$ — некоторый орграф, подобный одной компоненте связности управляющей граф-схемы (Q — множество его вершин, а X — множество его дуг), и пусть $\mathcal{G}' = conv(\mathcal{G}) = (Q', X')$, где $Q' = \{a' \mid a \in X\} \cup$ $\cup \{\alpha', \omega'\}$ — множество вершин графа $\mathcal{G}'$, причем вершина a' графа $\mathcal{G}'$ есть образ дуги a графа $\mathcal{G}$, α' и ω' — соответственно начальная и конечная вершины графа $\mathcal{G}'$, а X' — множество дуг графа $\mathcal{G}'$.

---

[146] Точнее, так получается прототип тестового варианта, представленный как цепочка лексических классов. Чтобы получить конкретный тестовый вариант, в этой цепочке необходимо заменить каждый лексический класс одним из его представителей.

Рассмотрим некоторый полный маршрут (т.е. маршрут из начальной вершины α' в конечную вершину ω') в графе $\mathcal{G}'$ $\mu' = \alpha'_1\chi'_1\beta'_1 \dots \beta'_m\chi'_{m+1}\omega'$. Он проходит через вершины α'_1, β'_1, … , β'_m, ω', причем дуга χ'_1 инцидентна вершинам α' и β'_1,…, дуга χ'_{m+1} инцидентна вершинам β'_m и ω'.

Прообразом маршрута μ' в графе $\mathcal{G}'$ назовем такой маршрут $\mu = \alpha\chi_1\beta_1 \dots \beta_m\chi_{m+1}$ в графе $\mathcal{G}$, в котором дуга χ_1 есть прообраз вершины β'_1, α — начало дуги χ_1 (начальная вершина графа $\mathcal{G}$), β_1 — конец дуги χ_1, ..., дуга χ_{m+1} — прообраз вершины β'_{m+1}, причем вершина β_m есть ее начало, а ω — конец (который также является конечной вершиной графа $\mathcal{G}$). Операцию нахождения прообраза маршрута μ' обозначим символом im^{-1}. Тогда запись $\mu = im^{-1}(\mu')$ обозначает, что маршрут μ в графе $\mathcal{G}$ есть прообраз маршрута μ' в графе $\mathcal{G}'$.

***Определение* 4.** Пусть $\mathcal{G}^n = \mathrm{conv}^n(\mathcal{G})$, и μ' — некоторый полный маршрут в $\mathcal{G}^n$. Положим по определению $im^0(\mu') = \mu'$, а для k, $1 \le k \le n$, будем считать, что $im^{-k}(\mu') = im(im^{-(k-1)}(\mu'))$. Маршрут $\mu = im^{-n}(\mu')$ назовем прообразом маршрута μ' в исходном графе $\mathcal{G}$.

Итак, общий порядок генерации оптимального теста уровня n, можно оформить в виде следующего алгоритма.

Алгоритм 9.1: генерация теста порядка *n*.

Вход: $\mathcal{G}$ — орграф, представляющий некоторую явнорегулярную граф-схему без резольверов; $n \ge 1$ — некоторое заданное натуральное значение.

Выход: T_n — тест уровня n, сгенерированный по $\mathcal{G}$.

Метод: Искомый тест строится по следующим шагам:

1. Строится орграф $\mathcal{G}^{n-1} = \mathrm{conv}^{n-1}(\mathcal{G})$.

2. Путем решения задачи китайского почтальона на орграфе $\mathcal{G}^{n-1}$ находится множество полных маршрутов M', обеспечивающих минимальное реберное покрытие орграфа $\mathcal{G}^{n-1}$.

3. Для каждого маршрута $\mu' \in M'$ строится его прообраз в $\mathcal{G}$

$$\mu = im^{-(n-1)}(\mu').$$

Множество всех таких прообразов обозначим через M.

4. Искомый тест T_n получается заменой лексических классов произвольными их представителями в порождениях[147] маршрутов из множества *M*.

***Замечание* 1.** Описанный алгоритм обеспечивает одно из возможных минимальных (по числу вызовов) покрытий множества допустимых комбинаций из k ($1 \le k \le n$) семантик.

***Замечание* 2.** Если исходный граф $\mathcal{G}$ не имеет циклов и петель, то при всех $n \ge 1$ множества T_n одинаковы, т.е. тесты всех уровней, построенные по нему, равны между собой.

***Замечание* 3.** Очевидно, что если тестирование уровня n ($n > 1$) проведено, нет смысла проводить тестирование уровня k при $k < n$, поскольку тест уровня n удовлетворяет критерию полноты порядка k.

***Замечание* 4.** В принципе, тесты генерируются по грамматике входного языка, представленной в форме графа, т.е. в конечном итоге по структуре входных данных программы, моделирующей соответствующий процессор. Эти тесты покрывают управляющую структуру программы процессора в такой же мере, как и структуру входных данных, поскольку главная часть логики такой программы сосредоточена в управляющей таблице процессора, которая генерируются по той же грамматике, что и тесты.

Другими словами, характерная особенность любой синтаксически управляемой программы состоит в том, что одна и та же спецификация определяет как структуру данных, обрабатываемых этой программой, так и структуру управления самой этой программы. Эта особенность благоприятствует проведению тестирования такого рода программ с полным учетом специфики их организации в отличие от методов тестирования, рассматривающих тестируемую программу как "черный ящик". И в самом деле, используя ту же спецификацию для генерации тестов, мы в полной мере учитываем наше знание об организации тестируемого средства синтаксически управляемой обработки данных.

В заключение этого раздела приведем пример, иллюстрирующий вышеизложенную технику генерации тестов[148].

Пример генерации теста уровня 3.

Пусть $\mathcal{G}$ — орграф, изображенный на рис. 9.1, *а*, и $n = 3$. Рис. 9.1, *б* представляет орграф $\mathcal{G}^1 = \mathrm{conv}(\mathcal{G})$, а рис. 9.1, *в* — орграф $\mathcal{G}^2 = \mathrm{conv}^2(\mathcal{G})$. Одно из возможных минимальных реберных покрытий графа $\mathcal{G}^2$ есть

$$M'' = \{(1)\,\alpha''\chi_1''\beta_1''\chi_2''\beta_2''\chi_3''\beta_3''\chi_4''\omega'',$$

$$(2)\,\alpha''\chi_5''\beta_4''\chi_7''\beta_7''\chi_{11}''\beta_9''\chi_{14}''\omega'',$$

$$(3)\,\alpha''\chi_5''\beta_4''\chi_6''\beta_5''\chi_8''\beta_6''\chi_{10}''\beta_9''\chi_{14}''\omega'',$$

$$(4)\,\alpha''\chi_5''\beta_4''\chi_6''\beta_5''\chi_9''\beta_8''\chi_{13}''\beta_8''\chi_{12}''\beta_6''\chi_{10}''\beta_9''\chi_{14}''\omega''\}.$$

---

[147] Под порождением маршрута подразумевается след от его вершин.

[148] Не раскрывая пока подробностей выполнения шага 2 алгоритма.

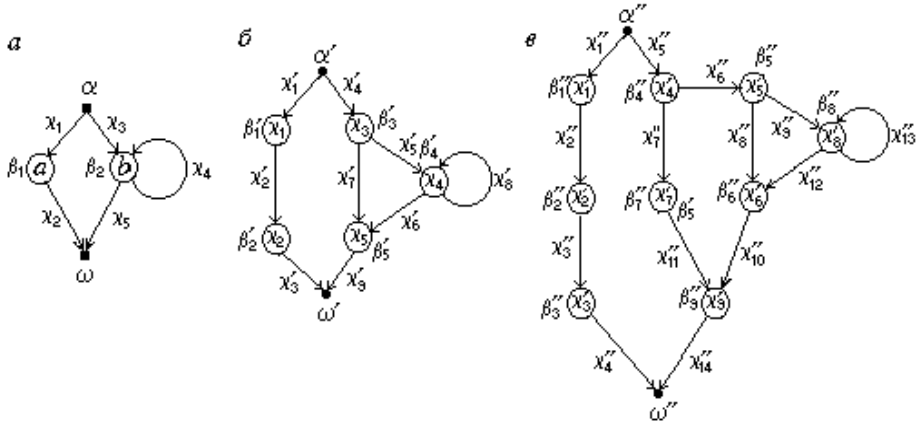


Рис. 9.1. Преобразования исходного орграфа
при генерации теста порядка 3.

Прообраз M'' в $\mathcal{G}^1$ есть

$$M' = \{\ (1)\ \alpha'\chi'_1\beta'_1\chi'_2\beta'_2\chi'_3\omega'\ ,$$
$$(2)\ \alpha'\chi'_4\beta'_3\chi'_7\beta'_5\chi'_9\omega'\ ,$$
$$(3)\ \alpha'\chi'_4\beta'_3\chi'_5\beta'_4\chi'_6\beta'_5\chi'_9\omega'\ ,$$
$$(4)\ \alpha'\chi'_4\beta'_3\chi'_5\beta'_4\chi'_8\beta'_4\chi'_8\beta'_4\chi'_6\beta'_5\chi'_9\omega'\ \}.$$

Прообраз M' в $\mathcal{G}$ есть

$$M = \{\ (1)\ \alpha\chi_1\beta_1\chi_2\omega,$$
$$(2)\ \alpha\chi_3\beta_2\chi_5\omega,$$
$$(3)\ \alpha\chi_3\beta_2\chi_4\beta_2\chi_5\omega,$$
$$(4)\ \alpha\chi_3\beta_2\chi_4\beta_2\chi_4\beta_2\chi_4\beta_2\chi_5\omega\}.$$

Искомый тест уровня 3 есть $T_3 = \{(1)\,a,\ (2)\,b,\ (3)\,bb,\ (4)\,bbbb\}$. Он состоит из четырех тестовых вариантов, активизирующих семантики, которые помечают дуги исходного графа $\mathcal{G}$ (см. рис. 9.1, *а*), во всевозможных сочетаниях вплоть до всех троек.

9.4. МЕТОД РЕШЕНИЯ ЗАДАЧИ О КИТАЙСКОМ ПОЧТАЛЬОНЕ

Как уже отмечалось, основная деталь генератора тестов (см. шаг 2 алгоритма 9.1) — алгоритм решения задачи о китайском почтальоне. Здесь мы уточним связь генерации оптимальных тестов с задачей о китайском почтальоне и опишем один из возможных методов ее решения. Как известно, эта задача состоит в нахождении кратчайшего замкнутого маршрута в графе, который включает каждую дугу по крайней мере один раз. Эта задача может ставиться как на неориентированном, так и на ориентированном графах, но не всегда имеет решение.

В нашем случае мы имеем дело с ориентированным графом, который обладает той благоприятной для существования решения особенностью, что каждая

его дуга входит по крайней мере в один маршрут из начальной в конечную вершину. Если в этом графе провести дополнительную дугу из конечной вершины в начальную, то мы получим ориентированный граф, на котором задача о китайском почтальоне всегда разрешима.

Предположим, что минимальный замкнутый маршрут, покрывающий все дуги такого графа, уже найден. Тем самым установлено, сколько раз каждая дуга орграфа входит в найденный маршрут. Эти числа назовем *кратностями дуг*.

Теперь преобразуем наш граф еще раз, проведя параллельно каждой дуге дополнительные равнонаправленные дуги так, чтобы их общее число равнялось кратности данной дуги. Полученный таким образом орграф назовем *пополненным*. Очевидно, что пополненный граф является эйлеровым: он содержит эйлеров цикл — замкнутую цепь, включающую все его вершины и дуги. Эта цепь и есть оптимальный маршрут китайского почтальона.

Таким образом задача генерации теста сводится к нахождению дополнительных кратностей дуг данного орграфа и нахождению эйлерова цикла в пополненном графе.

Нахождение дополнительных кратностей дуг производится, с учетом следующих условий:

1) кратность каждой дуги положительна;

2) для каждой вершины сумма кратностей дуг, входящих в нее, равна сумме кратностей дуг, выходящих из нее (необходимый и достаточный признак эйлеровости пополненного графа);

3) сумма дополнительных кратностей минимальна.

Пусть $\mathcal{G} = (Q, X)$ — данный орграф с дополнительной дугой из конечной в начальную вершину. Пусть $x, y \in Q$ — вершины, $(x,y) \in X$ — дуга из вершины x в вершину y, а $f(x,y)$ — дополнительная кратность дуги (x,y).

Обозначим через $d^-(x)$ число дуг, входящих в вершину x, а через $d^+(x)$ — число дуг, выходящих из нее. Тогда задача нахождения дополнительных кратностей дуг сводится к решению недоопределенной системы линейных алгебраических уравнений вида

$$\{d^-(x)\} + \sum_y f(y,x) = d^+(x) + \sum_y f(x,y)\}_{x \in Q}$$

или

$$\{\sum_y [f(x,y) - f(y,x)] = d^-(x) - d^+(x) = D(x)\}_{x \in Q}$$

относительно неизвестных $f(x,y)$ при дополнительных условиях:

$$\sum_{(x,y) \in X} f(x,y) = \min \text{ и } \{f(x,y) \geq 0\}_{(x,y) \in X}.$$

Другими словами, эта задача сводится к целочисленной задаче линейного программирования, представляющей одну из модификаций задачи о максимальном потоке минимальной стоимости. В потоковой терминологии $f(x,y)$ — поток, протекающий по дуге (x,y), длина которой, как и всех других дуг графа, в нашем случае равна 1.

Эту задачу решает алгоритм, приводимый далее.

Алгоритм 9.2: нахождение максимального потока минимальной стоимости.

Вход: $\mathcal{G} = (Q, X)$ — орграф с дугой из конечной вершины в начальную.

Выход: Поток $\{f(x,y)\}_{(x,y)\in X}$, такой, что $\sum_{(x,y)\in X} f(x,y) = \min$.

Метод: Алгоритм выполняется по следующим шагам:

1. Исключить петли из исходного орграфа $\mathcal{G}$ (поскольку каждая петля одновременно является входной и выходной дугой в некоторую вершину графа, ее удаление не изменяет отношения между числом входящих и числом выходящих дуг).

2. Определить источники и стоки: x — *источник*, если $D(x) > 0$; x — *сток*, если $D(x) < 0$.

3. Построить дополнительный *главный источник* S и провести из него дуги во все имеющиеся источники; построить дополнительный *главный сток* T и провести в него дуги из всех имеющихся стоков. Все дуги, инцидентные главному источнику или главному стоку назовем *крайними дугами*.

4. Задать пропускные способности дуг: $c(S,x) = D(x)$, если x — источник, $c(x,T) = -D(x)$, если x — сток. Пропускные способности всех других дуг не ограничены. Таким образом, теперь мы имеем дело с сетью.

5. Задать для начала на всех дугах сети нулевой поток: $f(x,y) := 0$.

6. Задать нулевые значения вершинных чисел для всех вершин сети: $p(x) := 0$.

7. "Окрасить" главный источник S.

8. Выполнить процедуру окрашивания вершин и дуг сети (см. алгоритм 9.3 далее).

9. Если главный сток T окрасился, то найти маршрут M из S в T, составленный из окрашенных дуг, и перейти к шагу 11. Иначе

10. Увеличить на единицу вершинные числа для всех неокрашенных вершин: $p(x) := p(x) + 1$, и перейти к шагу 8.

11. Определить величину изменения имеющегося потока вдоль маршрута M: $\Delta f = \min\{\min_{a\in I}(i(a)), \min_{a\in R}(r(a))\}$, где $i(a) = c(a) - f(a)$; $r(a) = f(a)$. При этом считается, что $a = (x,y) \in I$, если $(x,y) \in M$ и $f(x,y) < c(x,y)$; и $a = (x,y) \in R$, если $(y,x) \in \mathrm{M}$ и $f(x,y) > 0$.

12. Изменить поток на каждой дуге маршрута M: $f(x,y) := f(x,y) + \Delta f$, если дуга $(x,y) \in I$; $f(x,y) := f(x,y) - \Delta f$, если дуга $(x,y) \in R$.

13. Проверить насыщенность крайних дуг. Дуга (x,y) считается *насыщенной*, если $f(x,y) = c(x,y)$. Если не все крайние дуги насыщены, сбросить окраску всех вершин сети, кроме вершины S, и перейти к шагу 8. В противном случае алгоритм завершен. Полученный поток f является *максимальным потоком минимальной стоимости*.

Используемая на шаге 8 процедура окрашивания представляется следующим алгоритмом.

Алгоритм 9.3: окрашивание сети.

Вход: Сеть с заданным на ней потоком f, пропускными способностями дуг c и вершинными числами p.

Выход: Множество окрашенных вершин C_Q и множество окрашенных дуг C_X.

Метод: Алгоритм состоит из следующих шагов:

1. Положим для начала $C_Q := \{S\}$, $C_X := \varnothing$.

2. Над каждой дугой сети $(x, y) \notin C_X$ произведем следующие действия:

a) если $x \in C_Q, y \notin C_Q, p(y) - p(x) = 1, f(x, y) < c(x, y)$,

b) то $C_Q := C_Q \cup \{y\}$ и $C_X := C_X \cup \{(x, y)\}$;

c) если $x \notin C_Q$, $y \in CQ$, $p(y) - p(x) = 1$ и $f(x, y) > 0$,

d) то $C_Q := C_Q \cup \{x\}$ и $C_X := C_X \cup \{(x, y)\}$.

3. Шаг 2 повторять до тех пор, пока не будут просмотрены все дуги $(x, y) \notin C_X$.

По завершении этого алгоритма C_Q — множество окрашенных вершин, а C_X — множество окрашенных дуг.

После того, как в результате выполнения алгоритма 9.2 найден поток f, из сети удаляются главный источник и главный сток вместе с инцидентными им дугами, а удаленные на шаге 1 петли восстанавливаются. Полученная таким образом сеть с учетом дополнительных кратностей дуг (f) является *симметричной сетью*[149]. Именно в ней требуется найти эйлеров цикл. Для этого можно использовать нижеследующий алгоритм.

Алгоритм 9.4: нахождение эйлерова цикла.

Вход: Симметричная сеть.

Выход: C — искомый эйлеров цикл.

Метод: Предполагается, что на входной симметричной сети заданы кратности всех дуг $k(a) = f(a) + 1$, где $f(a)$ — поток, протекающий по дуге a.

Эйлеров цикл строится по следующим шагам:

1. Найти какой-нибудь цикл с началом в вершине α: $C := \text{cycle}(\alpha)$[150].

2. Если существуют вершина $x \in C$ и дуга (x, y), такая, что $k(x, y) > 0$, то перейти к шагу 3. Иначе алгоритм завершен.

3. Найти какой-нибудь цикл C_x с началом в вершине x: $C_x := \text{cycle}(x)$.

4. Присоединить цикл C_x к циклу C в вершине x: $C := \text{merge}(C, C_x)$[151] и перейти к шагу 2.

---

[149] *Симметричной сетью* называют такую, в каждой вершине которой имеет место баланс числа входных и выходных дуг.

[150] См. далее алгоритм 9.5.

[151] Очевидно, что функция merge не нуждается в дополнительных комментариях.

Используемая в этом алгоритме вспомогательная функция cycle, может быть реализована посредством следующего алгоритма.

Алгоритм 9.5: поиск цикла в сети.

Вход: Сеть и некоторая ее вершина x.

Выход: C—некоторый цикл с началом в заданной вершине x.

Метод: Предполагается, что заданы кратности всех дуг сети. Искомый цикл строится по следующим шагам:

1. Инициируется цепь C, содержащая единственную вершину x, которая запоминается как последняя вершина цепи: $l := x$.

2. Если в сети существует дуга (l,y) кратности $k(l,y) > 0$, то перейти к шагу 3. Иначе цикла, включающего вершину x, не существует — данная сеть не симметрична (но это не наш случай).

3. Присоединить вершину y к цепи C; уменьшить кратность дуги (l, y): $k(l, y) := k(l, y) - 1$; запомнить ее в качестве последней вершины цепи: $l := y$.

4. Если $l \neq x$, то перейти к шагу 2. Иначе алгоритм завершается, и C — искомый цикл.

Тестирование анализатора генераторов Алгола 68. В качестве примера приведем прототест первого порядка, сгенерированный по управляющей грамматике Gener (см. разд. 6.4). Он включает одиннадцать прототестовых вариантов для конструкции DECLARER (описатель) и два — для конструкции GENERATOR (генератор). Каждый из этих двух наборов прототестовых вариантов генерировался автономно по своей компоненте управляющей граф-схемы.

```
DECLARER-1: .struct ( DECLARER  tag , tag ,  DECLARER  tag )
DECLARER-2: .flex [ integer : integer , integer : integer , integer ] DECLARER
DECLARER-3: [ integer ] DECLARER
DECLARER-4: [ : , : ]  DECLARER
DECLARER-5: [ , , ] DECLARER
DECLARER-6: [ ] DECLARER
DECLARER-7: .proc ( DECLARER  ,  DECLARER ) DECLARER
DECLARER-8    : .proc DECLARER
DECLARER-9    : .union ( DECLARER  ,  DECLARER )
DECLARER-10   : .ref DECLARER
DECLARER-11   : ModeIndication
```

Длина теста = 59 Число вершин сети = 32

Время генерации 00:00:03 Число дуг сети = 66

```
GENERATOR-1: .heap DECLARER
GENERATOR-2: .loc DECLARER
```

Длина теста = 4 Число вершин сети = 5

Время генерации 00:00:00 Число дуг сети = 10

Суммарная длина прототестов (лексем) = 63

Для получения настоящего теста остается еще некоторым разумным образом "замкнуть" эти заготовки, т.е. превратить их в цепочки входного языка. Для этого необходимо заменить использующие вхождения имен конструкций в прототестовые варианты на экземпляры терминальных порождений, в которых лексические классы заменены подходящими представителями, и выполнить некоторую композицию из полученных таким образом терминальных цепочек.

Однако даже простой просмотр представленных прототестовых вариантов дает достаточную информацию для того, чтобы судить о правильности спецификации синтаксиса входного языка (упрощенных генераторов Алгола 68).

Замечание. Вероятно, возможно обобщение описанного здесь подхода к проблеме генерации оптимальных тестов на многокомпонентный вариант задачи о китайском почтальоне.

Что касается учета контекста[152] при такой генерации, то, по-видимому, одних только синтаксических средств для этого будет недостаточно. И хотя необходимость автоматической генерации выполнимых тестов заставляет обратиться к методам искусственного интеллекта, например таким, которые используются для верификации программ, думается что искусное применение контекстно чувствительных порождающих грамматик позволит преодолеть возникающие проблемы синтаксическими методами, но ценой проектирования и написания контекстно чувствительных грамматик, специально предназначенных для генерации выполнимых тестов.

[152] В приведенном примере прототеста фактических описателей массивов языка Алгол 68 тестовые варианты 4 – 6 контекстно ошибочны: позиции границ индексов в не могут быть пустыми.

ЗАКЛЮЧЕНИЕ
ИТОГИ И ПЕРСПЕКТИВЫ

Методология синтаксически управляемой обработки данных основывается на фундаментальных работах Н.Хомского по теории формальных грамматик, и, следовательно, ее история началась с середины 50-х годов ушедшего столетия. С другой стороны, за 20 лет до основополагающей работы Н. Хомского "Three models for the description of language" [9] А. М.Тьюрингом были созданы [19] предпосылки для развития теории автоматов — концептуальной основы реализации трансляций. Таким образом, эти два имени должны быть названы первыми среди основоположников методологии синтаксически управляемой обработки данных.

Первой областью, в которой синтаксические методы принесли практически значимые результаты, несомненно, является трансляция языков программирования. Все современные компиляторы языков программирования, обладающих нетривиальным синтаксисом, по существу, основаны на синтаксических методах перевода: сборка объектного кода в них производится по синтаксической структуре входной программы. В начале 60-х годов КС-грамматики с успехом были использованы для описания синтаксиса Алгола 60, и с тех пор этот класс грамматик используется для описания языков программирования или они применяются в качестве рабочих грамматик в системах автоматического построения трансляторов.

Среди первых отечественных трансляторов, которые использовали принцип синтаксического управления, следует назвать транслятор для Алгола 60, разработанный в начале 60-х годов в Отделе прикладной математики АН СССР коллективом программистов, возглавляемых проф. М. Р. Шура-Бура. Блок синтаксического анализа этого транслятора настраивался на конкретный входной язык с помощью специальной таблицы. Правда, эта таблица строилась вручную.

В конце 60-х годов у меня была возможность детально изучить реализацию Алгола 60 для датской вычислительной машины GIER в A/S Regnecentralen (Копенгаген). Каждый из девяти просмотров компилятора GIER ALGOL 4, написанного коллективом разработчиков под руководством проф. П. Наура, концептуально организован как магазинный автомат. Управляющие таблицы для этих просмотров строились вручную по грамматикам входного и промежуточных языков, используемых для обмена информацией между просмотрами. Именно этот проект послужил стимулом для поиска технологии автоматического построения языковых процессоров автоматного типа по спецификации языка и показал возможность синтеза прикладной программы из таких процессоров.

Примерно к тому же времени относится период наибольшей активности в этой области А. Л. Фуксмана из Ростовского государственного университета, внесшего значительный вклад в развитие синтаксических методов трансляции и технологию систем построения трансляторов (СПТ). В Ленинградском университете это направление с конца 60-х годов развивалось в связи с работами по реализации Алгола 68, выполнявшимися под научным руководством Г. С. Цейтина. В предварительном отчете по методам реализации системы программирования на базе Алгола 68 [1] синтаксическим проблемам была посвящена гл. 1.

С тех пор методология синтаксически управляемой обработки данных в Санкт-Петербургском университете продолжала развиваться, охватывая все более широкий круг задач, который в настоящее время включает: тестирование свойств грамматик, их эквивалентные преобразования, автоматическую генерацию языковых процессоров по спецификациям трансляций, оптимизацию языковых процессоров, автоматическую генерацию диагностических сообщений периода процессирования, многопроцессорную[153] обработку, генерацию оптимальных тестов и обустройство удобной среды для отладки изготовленных средств синтаксически управляемой обработки данных. Уточнялись и концептуальные основы методологии: сформировалась концепция регулярных сплайнов и сплайновых контекстно чувствительных челночных процессоров, определилась парадигма объектно-синтаксического программирования, сложилась архитектура многопроцессорных комплексов синтаксически управляемой обработки данных.

Сопоставляя SYNTAX-технологию с другими технологиями, такими, как YACC [13,16], GAG [14] или Eli [11], можно отметить следующие ее существенные особенности.

Синтаксическая основа SYNTAX-технологии базируется на использовании RBNF-грамматик, расширенных контекстными символами, интерпретируемыми как преобразования операционной среды и предикаты, тестирующие ее состояние. За ними стоят конечные и магазинные процессоры (одно-просмот-ровые или челночные, простые или контекстно чувствительные). Причем магазинные процессоры, используемые в SYNTAX-технологии, отличаются от классических тем, что, по существу, используют магазин лишь для сопряжения конечных процессоров, реализующих обработку КС-языка, аппроксимируемого регулярными фрагментами, образующими регулярный сплайн[154]. Это позволяет снизить затраты на обработку КС-языков почти до уровня конечно-автоматных.

---

[153] Имеется в виду комплексирование нескольких языковых процессоров в рамках одного SYNTAX-приложения.

[154] Именно за эту особенность они получили название *сплайновых процессоров*.

Подготовка спецификации трансляции в SYNTAX-технологии состоит в том, что в правила обычной КС-грамматики вносятся контекстные условия и семантика в виде контекстных символов (семантик и резольверов), с параллельным описанием интерпретирующих их процедур и логических функций, а затем полученная таким образом трансляционная грамматика подвергается эквивалентным преобразованиям с целью ее максимально возможной регуляризации. Разумеется, в результате этих преобразований упрощается синтаксическая структура входного языка, вплоть до того, что она полностью исчезает, если входной язык регулярен. Но она и не нужна после того, как в ней был запланирован контекст и семантика. Наоборот, чем меньше регулярных фрагментов остается в аппроксимации входного языка, тем меньше конечных автоматов используется для его обработки. В предельном случае, когда входной язык регулярен, мы имеем один регулярный фрагмент и один конечно-автоматный процессор для него.

Деформация исходной синтаксической структуры, конечно, не желательна для диагностики ошибок периода процессирования, поскольку диагностические сообщения об ошибках должны формулироваться в терминах исходной грамматики, тогда как фактически синтаксический анализ идет по другой — рабочей (регуляризированной) грамматике. Введение в SYNTAX-технологии концепции вспомогательных понятий и синонимов для них и нетерминалов практически полностью компенсирует исчезновение структурной информации из рабочей грамматики.

Технология YACC базируется на кнутовских LALR(1)-грамматиках. Две другие системы основаны на атрибутных грамматиках Д. Кнута. Бесконтекстная основа технологии Eli является открытой[155].

Синтаксический анализ в SYNTAX-технологии накладывает ограничения на класс грамматик, характерные для методов анализа сверху вниз, тогда как LALR-грамматики ориентированы на анализ по схеме снизу вверх.

Атрибутная техника, используемая в GAG и Eli, предполагает [14] построение бесконтекстного дерева вывода входной цепочки, и затем внесение в него контекстной информации[156] путем вычисления значений атрибутов символов, помечающих его узлы. В SYNTAX-технологии учет контекстной информации синхронизирован с просмотром входа, и дерево вывода строится с учетом контекстной информации так, что его структура зависит от нее.

Интересной особенностью SYNTAX-технологии является и то, что наряду с генерацией анализирующих процессоров она позволяет генерировать также и порождающие процессоры и сочетать те и другие в рамках одного SYNTAX-приложения. Именно эта ее особенность позволяет строить программы, в которых описание и реализация управляющей структуры алгоритма обработки

[155] И вообще, Eli является открытой системой в том смысле, что ее компоненты могут легко заменяться другими функционально равнозначными и без особых хлопот в нее могут добавляться новые функциональные блоки.

[156] Хотя фактическое вычисление атрибутов может производиться по ходу построения дерева вывода, это не может изменить того факта, что структура такого дерева определяется без учета контекстной информации.

данных отделены от описания самих данных и процедур их обработки. Такая архитектура программ четко локализует их свойства, благодаря чему они лучше поддаются модификации при минимальных дополнительных расходах на перекомпиляцию.

За годы работы над совершенствованием SYNTAX-технологии постепенно накапливался также и опыт использования ее в практических разработках, таких, как серия компиляторов для Алгола 68, Паскаля, Ады, конвертор Reduce 2 → Mathematica, а также при построении самой технологической системы (преобразователь управляющих грамматик в форму управляющих граф-схем, реализация языка описания микролексики).

В течение нескольких лет SYNTAX-технология изучалась студентами кафедры математического обеспечения ЭВМ (ныне кафедры информатики) на математико-механическом факультете СПбГУ. Использование технологического комплекса SYNTAX в учебном процессе оказалось полезно не только студентам для практического овладения методами синтаксически управляемой обработки данных, но также помогло разработчикам в определении направления дальнейшего совершенствования технологического комплекса.

Фактически к настоящему времени завершается перенос технологического комплекса Syntax под операционную систему MS Windows. Студентка Е. А. Вергизова выполнила перенос подсистемы проектирования (за исключением генератора и редактора диагностических сообщений периода процессирования). Генератор диагностических сообщений перенес студент С.А.Гришин. Он же разработал многооконную версию редактора спецификаций. Расширяются функциональные возможности ТК Syntax. В частности, в него включена компонента визуализации управляющих граф-схем в виде синтаксических диаграмм Вирта, написанная аспирантом кафедры информатики И. Н. Наумовым[157].

Изначально в SYNTAX-технологии атрибуты не были предусмотрены. Использование семантических процедур и предикатных функций без параметров означает, что весь обмен контекстной информацией может осуществляться лишь через глобальную операционную среду. Это выводит разработку операционной среды из-под контроля технологии. Введение атрибутов в SYNTAX-технологию равносильно введению параметризации упомянутых семантических процедур и предикатных функций (резольверов). Разработка механизма поддержки атрибутов в данной технологии предаст ей большую надежность, а пользователю больше удобств и дополнительной информации при проектировании средств синтаксической обработки данных. Эта тема разрабатывается аспирантом А. С. Лукичёвым.

Автором разрабатывается программная поддержка эквивалентных преобразований спецификаций трансляций (регуляризация) и редукция КС-грамматик.

[157] Пример использования этой компоненты приведен на рис. 1.6 и 1.7.

ПРИЛОЖЕНИЕ

ТЕХНОЛОГИЧЕСКИЙ КОМПЛЕКС SYNTAX

1. НАЗНАЧЕНИЕ И ВОЗМОЖНОСТИ ТЕХНОЛОГИЧЕСКОГО КОМПЛЕКСА SYNTAX

Технологический комплекс (ТК) SYNTAX является интегрированной инструментальной системой, предназначенной для проектирования, разработки, реализации и тестирования средств синтаксически управляемой обработки данных. Типичные области его применения — лексический и синтаксический анализ языков программирования, автоматическое восприятие и преобразование языков с нетривиальной синтаксической структурой любые другие средства, предназначенные для этих целей (интерпретаторы и компиляторы, синтаксические редакторы, конверторы). Кроме того, он поддерживает методологию объектно-синтаксического программирования, предполагающую архитектуру программ со спецификацией данных и средств их обработки (объектов) отдельно от их управляющей структуры, описываемой посредством управляющей RBNF-грамматики.

Задача пользователя ТК — определить, какую трансляцию он желает реализовать, и написать соответствующую TSL-спецификацию. Назначение ТК — обеспечить соответствующие средства ее реализации.

Элементы, на базе которых строится реализация, — сплайновые процессоры. Они в значительной степени подобны классическим конечным и магазинным автоматам, но в отличие от них наделены чувствительностью к контексту, существенно расширяющей класс входных языков. Именно это свойство процессоров позволяет использовать их в качестве внутренней формы организации программ, внешнюю архитектуру которых можно описать метафорической формулой вида "Программа = Объекты + Грамматика".

Разрабатываемые с помощью ТК средства обладают способностью обнаруживать и диагностировать ошибки во входных данных в терминах той грамматики, при помощи которой определялся входной язык.

Технологический комплекс предоставляет собой интегрированную среду разработки с традиционным для персональных компьютеров оконным интерфейсом пользователя, обеспечивающую:

- подготовку и редактирование TSL-спецификаций,
- проверку формальной правильности TSL-спецификаций,
- эквивалентные преобразования управляющих грамматик,
- преобразование управляющих RBNF-грамматик в форму управляющих граф-схем,
- построение управляющих таблиц процессоров,
- оптимизацию управляющих таблиц процессоров,
- генерацию диагностических сообщений периода процессирования,
- редактирование диагностических сообщений "на фоне" управляющей грамматики,
- генерацию тестов (прототестов) для проверки адекватности полученного средства СУОД[158] потребностям заказчика,
- комплексирование нескольких языковых процессоров в единое средство СУОД,
- трассировку работы средств СУОД,
- раскрытие синтаксической структуры входной цепочки.

Технологический комплекс SYNTAX состоит из подсистемы проектирования (Designing Subsystem) и подсистемы процессирования (Processing Subsystem). В первой подсистеме решаются все синтаксические задачи проектирования (генерация управляющих граф-схем, таблиц, диагностик и тестов), во второй — микролексические (генерация словарей для штатного транслитератора) и семантические (компиляция операционной среды). В режиме процессирования в ней можно трассировать работу построенного средства СУОД.

В любой момент работы с ТК SYNTAX доступна контекстно зависимая справка, вызываемая на экран нажатием клавиши F1. Справочная система[159] реализована в виде двух гипертекстовых документов, относящихся к соответствующим подсистемам, и может служить интерактивным справочником по всему комплексу.

2. ПОДСИСТЕМА ПРОЕКТИРОВАНИЯ

Подсистема проектирования средств СУОД состоит из нескольких функциональных блоков, которые поддерживают выполнение задач проектирования, относящихся к синтаксическому уровню реализации трансляции. Инструментарий подсистемы проектирования становится доступен для использования после ее запуска посредством традиционного для персональных компьютеров интерфейса в стиле Борланд.

---

[158] СУОД — **С**интаксически **У**правляемой **О**бработки **Д**анных.

[159] Существует другой (автономный) справочник, доступный по команде WinHelp. Он содержит более общую информацию о SYNTAX-технологии.

Запуск подсистемы производится по команде **Design**[*Name*], где *Name* — имя файла, содержащего TSL-спецификацию[160]. В результате на экране появляется заставка подсистемы проектирования, представленная на рис. П.1.

Рис. П.1. Вид заставки подсистемы проектирования.

После ее сброса клавишей **Esc**, открывается панель подсистемы проектирования с главным меню вверху и подсказками внизу. Эта панель пуста, если имя TSL-спецификации не было задано (рис. П.2), или в нее вставлено окно редактора с указанной TSL-спецификацией, если при запуске подсистемы проектирования так или иначе было задано имя файла TSL-спецификации (рис. П.3). В этом случае можно немедленно начать ее редактирование или переходить к выполнению других операций.

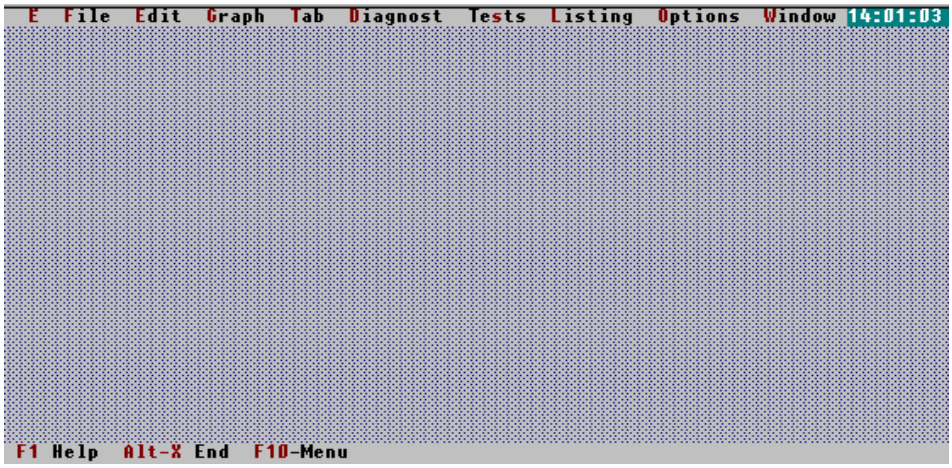


Рис. П.2. Вид панели подсистемы проектирования.

[160] По умолчанию подразумевается расширение .grm. В противном случае расширение должно быть указано явно.

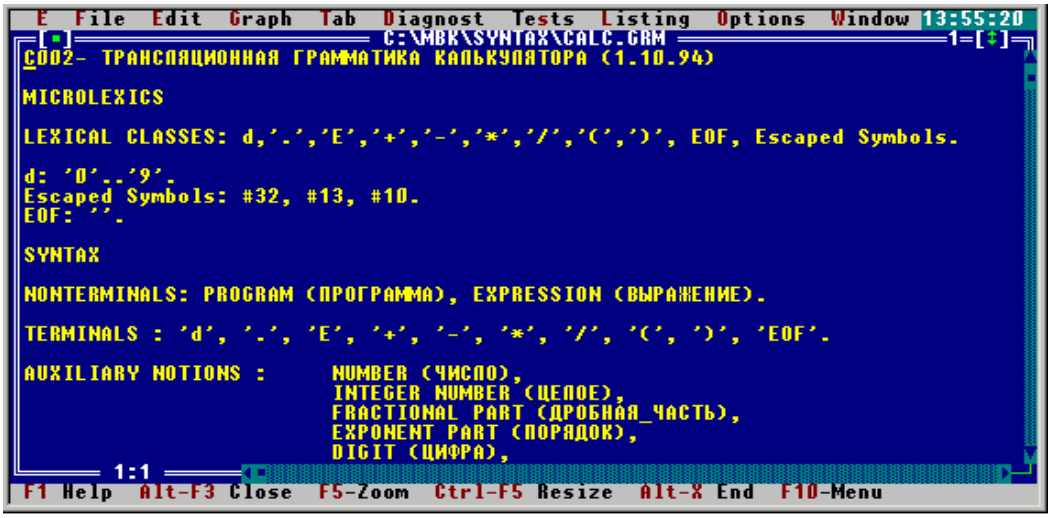


Рис. П.3. Вид окна редактора с загруженной TSL-спецификацией.

Функции подсистемы. Здесь мы только перечислим функции, представленные в главном меню подсистемы проектирования:

≅ Выдача справки о текущей версии ТК SYNTAX (рис. П.4).

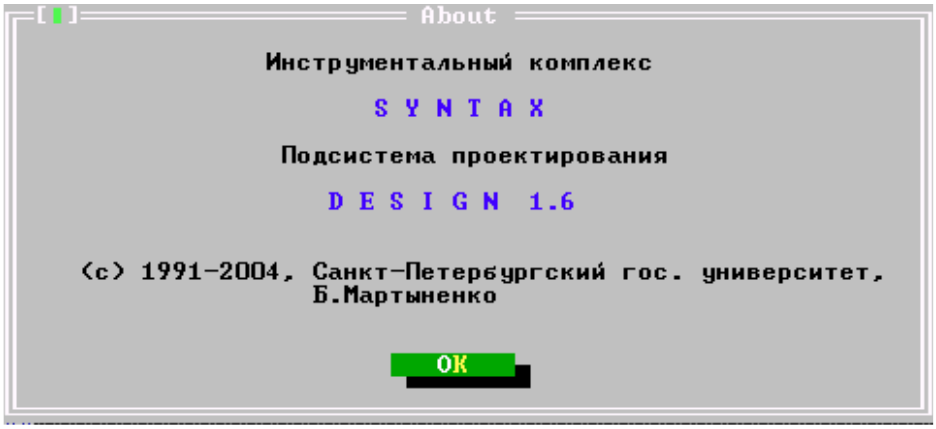


Рис. П.4. Вид справки о версии ТК SYNTAX.

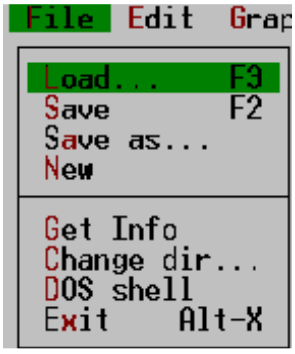


Рис. П.5. Меню File.

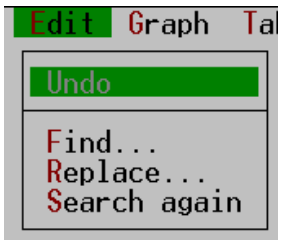


Рис.П.6. Меню Edit.

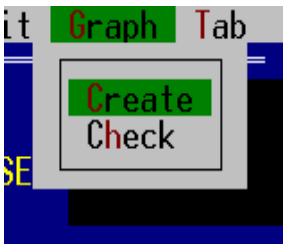


Рис.П.7. Меню Graph.

File Открывает системное выпадающее меню (рис. П.5), которое содержит следующие команды:

Load Загрузка файла в окно редактора. При выполнении этой команды открывается диалоговое окно, предоставляющее выбор файлов из текущего каталога с расширениями .grm.

Save Сохранение на диске спецификации, находящейся в окне редактора, под текущим именем (если текущее имя не задано, эта команда эквивалентна команде Save as).

Save as Сохранение на диске спецификации, находящейся в окне редактора, под именем, заданным в диалоговом окне этой команды.

New Открывает окно для написания новой TSL-спецификации.

Get Info Открывает окно со следующей информацией[161]:

имя главной спецификации,
каталог управляющих граф-схем,
каталог управляющих таблиц,
каталог диагностик,
каталог прототестов,
каталог листингов,
свободная оперативная память.

Change dir Назначение текущего каталога.

DOS shell Выход в среду DOS (возврат в среду ТК по команде **exit**).

Exit Выход из подсистемы проектирования.

Edit Открывает подменю (рис.П.6), которое содержит следующие команды:

Undo Отмена последнего изменения.

Find Поиск ближайшего вхождения образца.

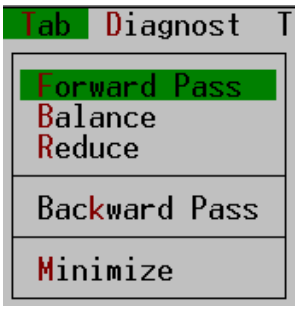
Replace Замена текущего вхождения заданного образца заданным текстом.

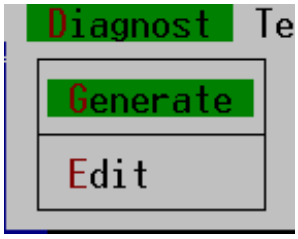
Search again Повторный поиск следующего вхождения образца.

Graph Открывает выпадающее подменю (рис.П.7), которое содержит следующие команды:

Create Генерация управляющей граф-схемы по управляющей RBNF-грамматике.

Check Проверка приведенности управляющей RBNF-грамматики[162].





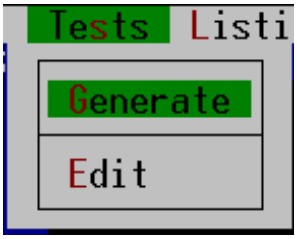


Рис.П.8. Меню Tab. Рис.П.9.Меню Diagnost. Рис.П.10. Меню Tests

Tab Открывает выпадающее подменю (рис.П.8), которое содержит следующие команды:

Forward pass Генерация управляющей таблицы прямого просмотра.

Balance Проверка внешней балансировки таблицы прямого просмотра.

Reduce Исключение левосторонней рекурсии из управляющей КС-грамматики.

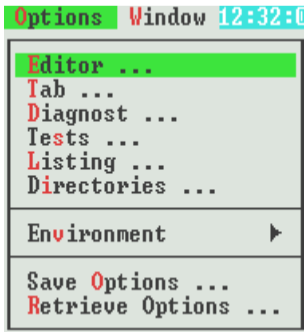
Backward pass Генерация управляющей таблицы обратного просмотра.

Minimize Оптимизация управляющей таблицы процессора.

---

161 Показанные установки могут быть изменены в том же окне.

162 Точнее, проверяется, во-первых, что каждый терминальный или нетерминальный символ грамматики выводим из начального нетерминала грамматики; во-вторых, что из каждого нетерминала выводится по крайней мере одна (хотя бы пустая) терминальная цепочка.



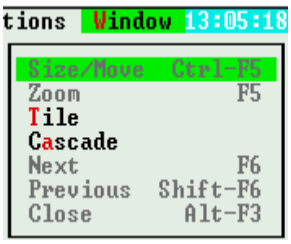


Рис. П.11. Меню Listing. Рис. П.12. Меню Options. Рис. П.13. Меню Window.

Diagnost Открывает выпадающее подменю (рис. П.9), которое содержит следующие команды:

Generate Генерация диагностических сообщений[163] периода процессирования. При этом используется шаблон, заданный по команде **Options / Diagnost**.

Edit Редактирование диагностических сообщений "на фоне" управляющей грамматики.

Tests Открывает выпадающее подменю (рис. П.10), которое содержит следующие команды:

Generate Генерация (прото-) теста. В открывающемся диалоговом окне отмечаются те конструкции, для которых требуется генерировать тесты (прототесты).

Edit Редактор теста[164] (поддерживает доработку прототеста до исполнимого теста).

Listing Открывает выпадающее подменю (рис. П.11), которое содержит следующие команды:

Create Генерация листинга, содержащего документы по процессору. Состав листинга и направление его выдачи (в файл или на принтер) задается при помощи команды **Options / Listing**.

View Просмотр листинга на экране компьютера. Эта функция выполнима, если листинг направлялся в файл, а не на принтер.

Options Открывает выпадающее подменю (рис. П.12), которое содержит следующие команды, позволяющие производить настройку подсистемы проектирования:

Editor Настройка редактора (сохранять / не сохранять исходную версию редактируемого файла, использовать / не использовать режим AutoIndent).

Tab (В настоящее время не реализована).

Diagnost Настройка генератора диагностических сообщений (установка рамки формата диагностических сообщений, включение или выключение генерации диагностик для подавляемых состояний).

Tests Настройка генератора тестов (установка порядка теста, выбор режима выдачи контекстных символов в листинг теста).

Listing Настройка генератора листинга: выбор документов, включаемых в листинг; задание числа строк на странице; выбор направления выдачи листинга (в файл или на принтер).

---

163 Для оптимизированной версии процессора, если оптимизированные управляющие таблицы построены, а в противном случае — для обычной версии.

164 Пока не реализован.

Directories Задание каталогов (Directories), в которых сохраняются файлы с управляющими граф-схемами, управляющими таблицами, диагностиками, (прото-) тестами, листингами.

Environment Выбор предпочтений (Preferences), режимов экрана (Screen Mode) и мышки (Mouse).

Save Options Сохранение установленных режимов.

Retrieve Options Восстановление ранее сохраненных режимов.

Window Открывает выпадающее подменю (рис. П.13), которое содержит следующие команды:

Size / Move Изменение размеров окна и его перемещение по экрану.

Zoom Максимальное раскрытие окна.

Tile Мозаичное выстраивание окон.

Cascade Каскадное выстраивание окон.

Next Переключение на следующее окно.

Previous Переключение на предыдущее окно.

Close Закрытие активного окна.

Далее кратко описываются функциональные компоненты подсистемы проектирования.

Редактор TSL-спецификаций предназначен для их подготовки, просмотра и модификации[165]. Он готов к работе, как только в панели подсистемы открыто окно редактора, пустое (по команде **File/New**) — для набора новой спецификации, или содержащее TSL-спецификацию (загруженную по команде **File/Load**), которую требуется изменить.

Для одновременной работы над несколькими спецификациями можно открыть несколько окон редактора. При этом все операции, инициируемые командами меню, относятся к TSL-спецификации, находящейся в активном окне (активное окно выделено "двойной" рамкой), а переключение между окнами осуществляется посредством команд из раздела Window основного меню.

Имеются обычные возможности для редактирования, такие, как вставка, замена или удаление символов, разбиение и слияние строк, удаление, перемещение и копирование фрагментов текста, поиск и замена заданного текста. Есть возможность отмены последнего исправления.

Содержимое окна редактора может быть сохранено в любой момент, когда оно активно, под текущим (по команде **Save**) или новым (по команде **Save as …**) именем со стандартным расширением .grm или с произвольным другим расширением[166].

Генератор управляющих граф-схем служит для преобразования TSL-спецификаций в форму управляющих граф-схем и для проверки приведенности управляющей грамматики. Он запускается при помощи команды **Create**, подменю Graph главного меню подсистемы проектирования или последовательностью комбинации клавиш Alt+G и клавиши C, когда в окне редактора

---

165 Фактически редактор можно использовать для редактирования любых текстовых (ASCII) файлов.

166 Рекомендуется использовать стандартное расширение .grm, так как во многих случаях это избавит пользователя от необходимости указывать его явно.

находится соответствующая TSL-спецификация. Все время, пока выполняется функция Graph/Create, на экране находится окно статуса с заголовком Graph (рис. П.14), показывающее, какая часть задачи выполнена на текущий момент.

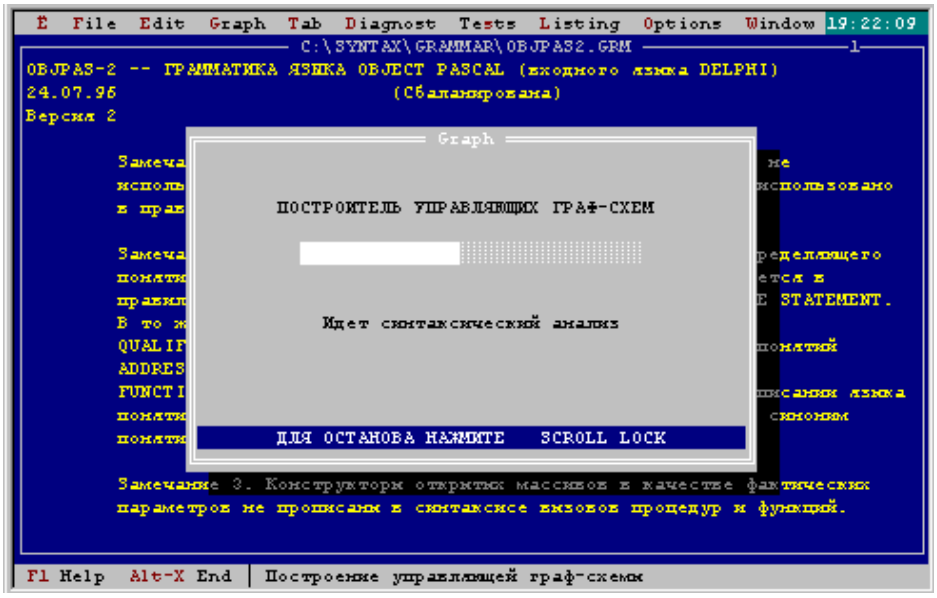


Рис. П.14. Вид окна технологической компоненты Graph
во время выполнения команды **Create**.

Генератор управляющих граф-схем анализирует текущую TSL-спецификацию и в случае обнаружения формальной ошибки выдает краткое диагностическое сообщение в специальной (красной) строке ошибок в верхней части экрана (рис. П.15).

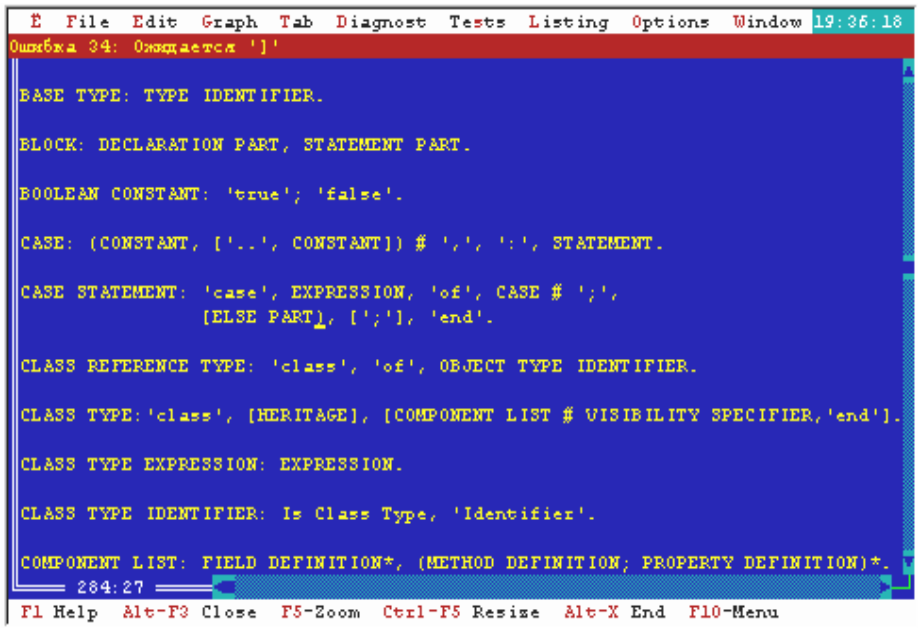


Рис. П.15. Вид окна технологической компоненты Graph с диагностическим сообщением о бесконтекстной ошибке.

При этом курсор указывает место ошибки в TSL-спецификации (строка 6 TSL-спецификации на рис. П.15). В это время по клавише F1 можно получить контекстную справку об ошибке (рис. П.16).

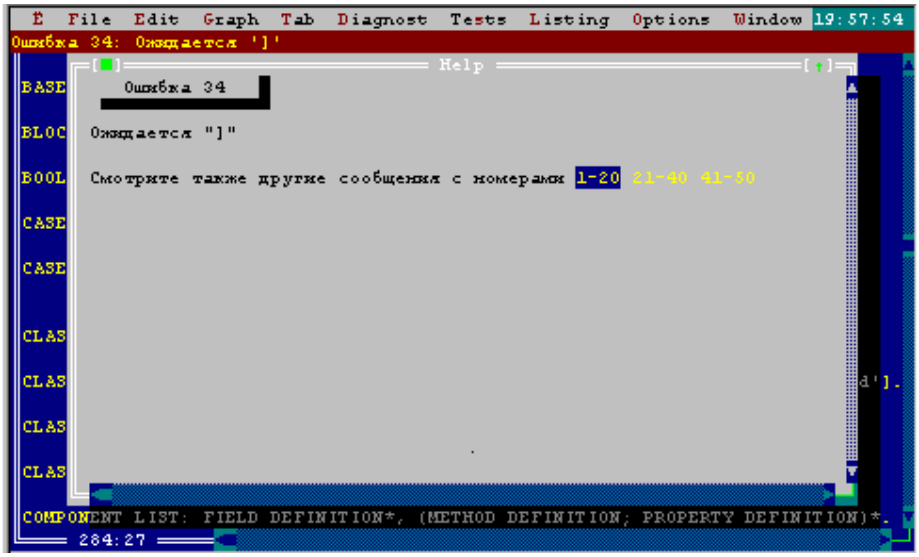


Рис. П.16. Вид окна технологической компоненты Graph
с контекстной справкой об ошибке.

При благополучном исходе, о котором сообщается сигналом Success в окне Graph, результатом является файл с текущим именем спецификации и расширением .grh, содержащий двоичный код управляющей граф-схемы.

После этого рекомендуется проводить дополнительную проверку граф-схемы на приведенность, используя команду **Graph / Check**. Недостижимые символы, обнаруженные в каком-либо из четырех словарей грамматики, нетерминалы, для которых не существуют определяющие их правила, а также нетерминалы, порождающие пустые языки, указываются в открывающемся в таком случае окне сообщений (рис. П.17).

Генератор управляющих таблиц предназначен для построения управляющих таблиц управляющих процессоров по управляющим граф-схемам. Он может быть запущен, когда в окне редактора находится соответствующая TSL- спецификация и управляющая граф-схема для нее построена.

Фактически генератор управляющих таблиц обладает четырьмя функциями, вынесенными в подменю компоненты Tab. В зависимости от того, предполагается ли однопросмотровая или челночная обработка, использование первоначальных или оптимизированных управляющих таблиц, в этом подменю выбираются команды **Forward pass** — для построения управляющих таблиц прямого просмотра, **Balance** — для проверки адекватности построенной таблицы прямого просмотра, **Backward pass** — для построения управляющей таблицы обратного просмотра, **Minimize** — для оптимизации управляющей таблицы только прямого просмотра (если таблица обратного просмотра не строилась) или для совместной оптимизации таблиц прямого и обратного просмотров (если они обе построены).

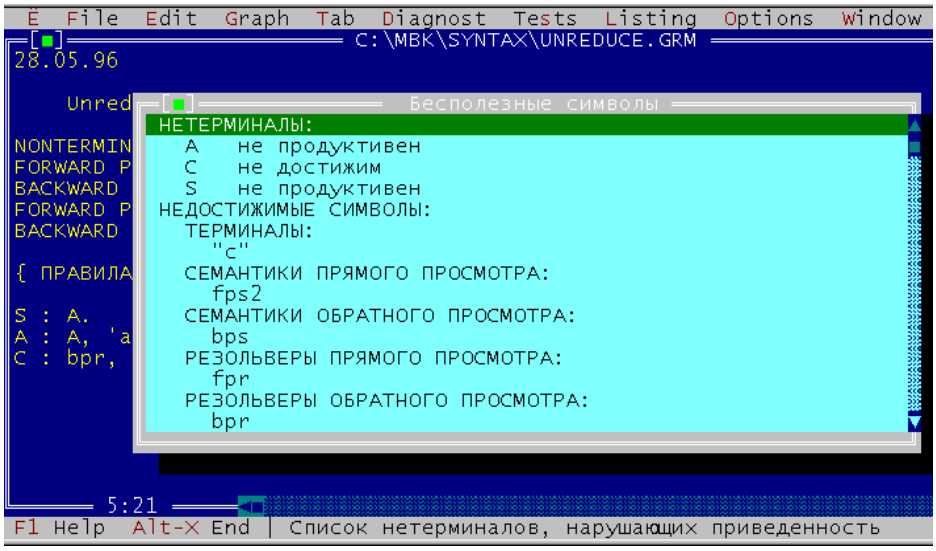


Рис. П.17. Вид окна технологической компоненты Graph после выполнения команды **Graph / Check**, обнаружившей неприведенность грамматики.

Функция Forward pass. При выполнении этой функции производится проверка соблюдения следующих ограничений, накладываемых на управляющую грамматику (граф-схему):

1) недопустимость использования начального нетерминала в правой части какого-либо правила;
2) недопустимость левосторонней рекурсивности;
3) недопустимость нетерминалов, которые не используются ни в одном выводе;
4) недопустимость непродуктивных нетерминалов (из каждого нетерминала должна выводится по крайней мере одна терминальная, хотя бы пустая, цепочка;
5) недопустимость непродуктивных петель в каждой компоненте графа;
6) синтаксическая балансировка;
7) семантическая балансировка.

Некоторые из этих проверок выполняются во время предварительного анализа управляющей граф-схемы, другие — по ходу построения управляющей таблицы. При обнаружении нарушения требований, предъявляемых к управляющей грамматике, на экране появляется окно, содержащее диагностическое сообщение об обнаруженном дефекте.

Процесс построения управляющей таблицы прямого просмотра состоит из нескольких этапов. О каждом из них пользователь извещается специальным окном статуса, в котором находится индикация степени завершенности соответствующей подзадачи (как, например, показано на рис. П.18). Наиболее тяжелый этап — разложение состояний, может быть приостановлен в любой момент нажатием клавиши Scroll Lock. В этом случае создается файл с текущим именем и расширением .fpd. Этот файл используется для продолжения работы в другое время.

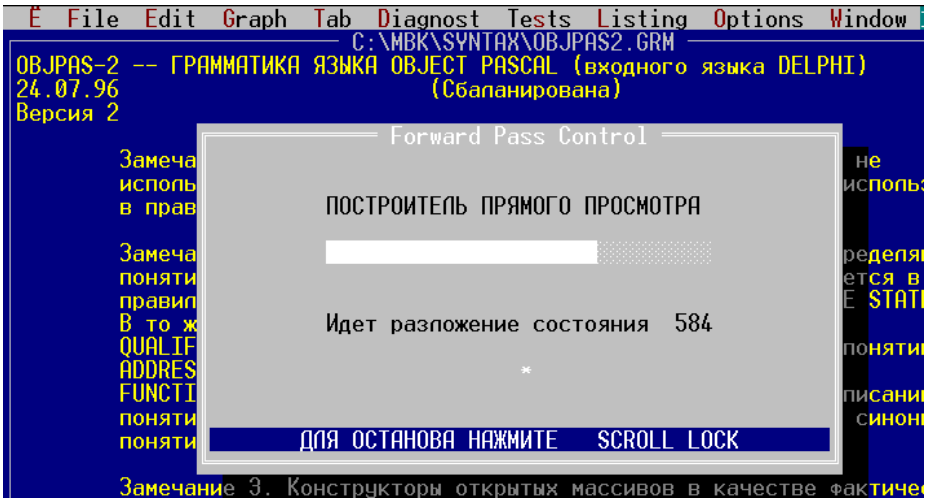


Рис. П.18. Вид окна технологической компоненты Tab
во время выполнения команды **Forward pass**.

Если никакие из перечисленных требований не нарушены, то в конце процесса образуется файл с текущим именем и расширением .fp, содержащий двоичный код управляющей таблицы прямого просмотра. Об успешном завершении процесса пользователь оповещается сигналом Success, появляющимся в окне статуса.

Если прямой просмотр реализуется сплайновым, а не конечным процессором[167], то построенная управляющая таблица подвергается последней проверке при помощи технологической функции **B**alance.

Функция Balance. Содержательный смысл последней проверки[168], осуществляемой этой функцией, состоит в том, чтобы гарантировать детерминизм сплайнового процессора прямого просмотра относительно выбора между движением, использующим входной символ, и ε-движением. В терминологии синтаксической структуры предложений входного языка это соответствует возможности однозначного отнесения каждого данного вхождения входного символа только к конструкциям одного и того же уровня.

Разумеется, выполнение команды **Balance** осмысленно, когда все предыдущие этапы успешно завершены, т.е. получен файл с расширением .fp.

После подачи команды **Balance** появляется диалоговое окно (рис. П.19) с запросом номера состояния, с которого требуется начать проверку внешней балансировки. После указания желательного номера[169] (по умолчанию установлено значение 1) необходимо нажать клавишу Enter.

---

[167] Попытка выполнить команду **Balance** для явнорегулярной грамматики приводит к появлению окна с сообщением о неуместности применения этой команды.

[168] Проверяется условие, называемое *внешней балансировкой*.

[169] Задавать номер, отличный от 1, необходимо только в том случае, если требуется продолжить проверку, прерванную ранее нажатием клавиши Scroll Lock.

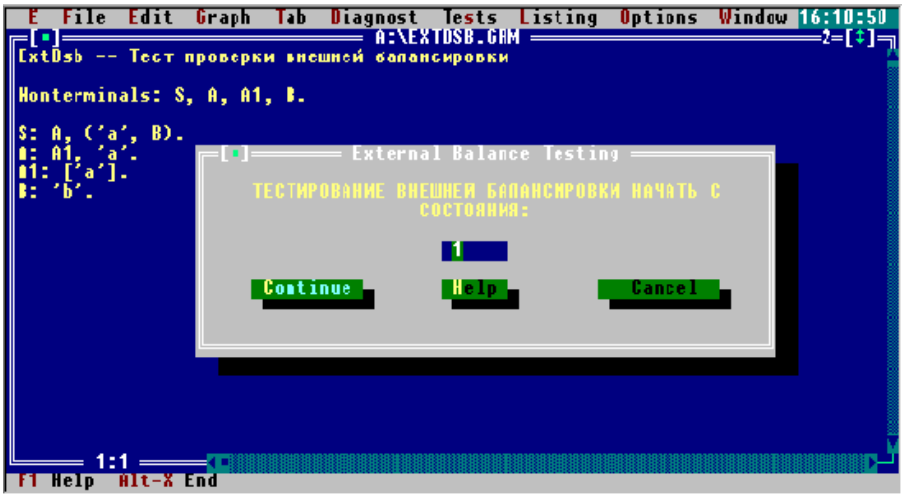


Рис.П.19. Вид диалогового окна команды **Balance**.

Во время исполнения команды **Balance** на экране находится окно статуса с индикацией степени завершенности процесса проверки и номером состояния, для которого идет проверка в текущий момент. Об успешности проверки пользователь извещается сигналом Success в окне статуса. В случае обнаружения внешнего дисбаланса появляется специальное окно (рис. П.20) с сообщением о том, в каких состояниях и по каким входным символам нарушена внешняя балансировка. Все такие случаи можно просмотреть, нажимая кнопку Continue, или отказаться от их просмотра, нажав кнопку Terminate. По кнопке Help можно получить контекстную справку (рис. П.21).

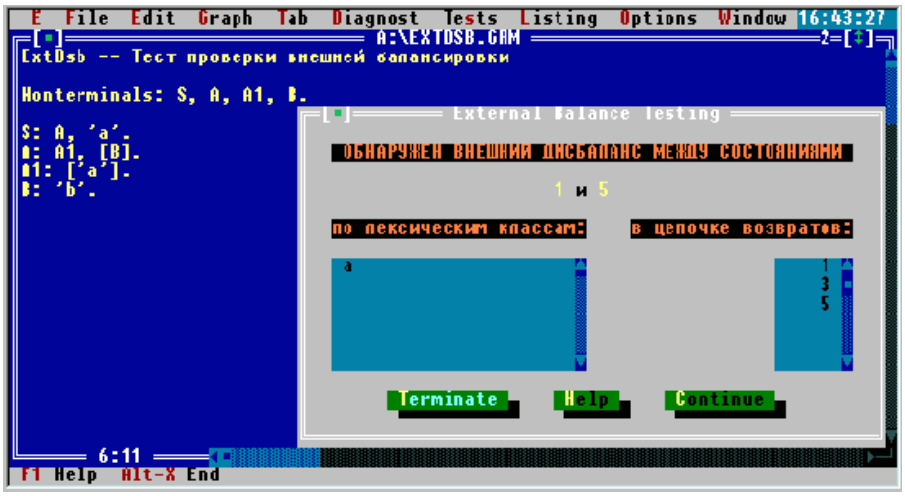


Рис. П.20. Вид сообщения о внешнем дисбалансе.

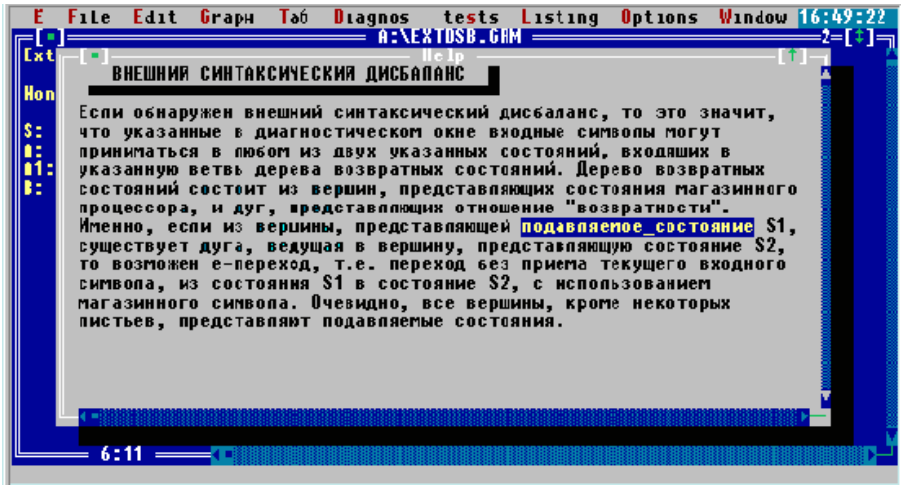


Рис. П.21. Вид контекстной справки о внешнем дисбалансе.

Наконец, после закрытия окна с сообщением о внешнем дисбалансе появляется предупреждение о деффектности построенных управляющих таблиц (рис. П.22).

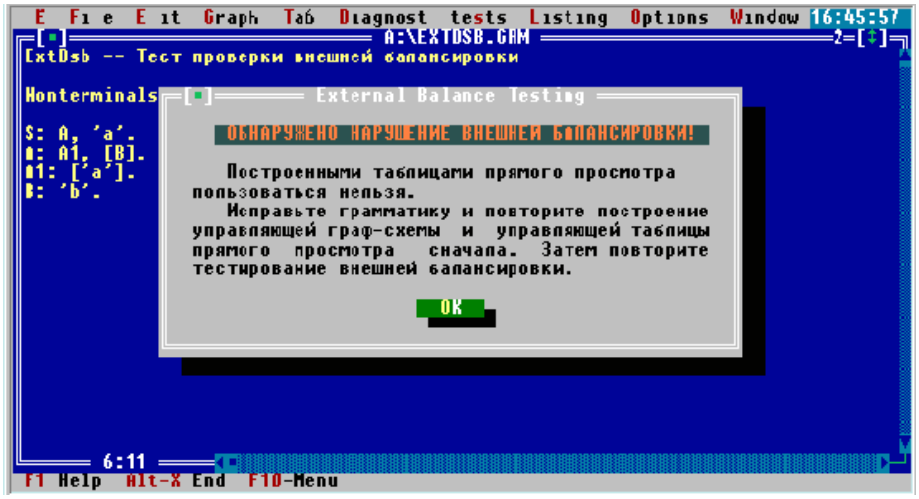


Рис.П.22. Окно с предупреждением о деффектности таблиц.

Разумеется, управляющая таблица прямого просмотра может использоваться для процессирования (и ее имеет смысл оптимизировать) только тогда, когда функция **Balance** не обнаруживает вышеупомянутого дефекта.

Функция Backward pass реализует построение управляющей таблицы обратного просмотра. Это необходимо, если предполагается челночная обработка (в управляющей грамматике имеются семантики обратного просмотра).

Во время выполнения этой команды на экране находится окно статуса с индикацией степени завершенности процесса. По ходу построения этой таблицы проверяется семантическая однозначность входного языка относительно семантик обратного просмотра. Если эта проверка проходит удачно, о чем Поль-

зователь извещается сигналом Success в окне статуса, на выходе образуется файл с текущим именем и расширением .bp, содержащим двоичный код управляющей таблицы обратного просмотра. В противном случае в окне сообщений выдается диагностика обнаруженной ошибки.

Функция Minimize выполняет оптимизацию управляющей таблицы простого или челночного процессора в зависимости от того, построена ли только таблица прямого просмотра[170] или построена также и управляющая таблица обратного просмотра[171]. В первом случае появляется диалоговое окно с предупреждением о том, что управляющая таблица обратного просмотра не существует, и запросом, требуется ли оптимизировать один только прямой просмотр.

Наиболее тяжелым этапом оптимизации является построение отношений эквивалентности на множествах состояний и магазинных символов. Поэтому предусмотрены возможности прерывания этого процесса и его продолжения в другое время. Для прерывания следует нажать клавишу Scroll Lock. В этом случае выдается файл с текущим именем и расширением .fpr, который содержит частично построенные отношения. Он используется при возобновлении процесса оптимизации[172].

После того, как процесс оптимизации благополучно завершается, о чем пользователь извещается сигналом Success в окне статуса, в текущем каталоге (или другом каталоге, выбранном для управляющих таблиц) появляются файлы с текущим именем и расширениями .mfp и .fcl, содержащие оптимизированную управляющую таблицу и классы эквивалентности входных символов, состояний и магазинных символов прямого просмотра, а если оптимизировались управляющие таблицы челночного процессора, то появляются также и одноименные файлы с расширениями .mbp и .bcl, содержащие оптимизированную управляющую таблицу и классы эквивалентности входных символов и состояний обратного просмотра.

Генератор диагностических сообщений автоматически готовит коллекцию диагностик, которые используются во время процессирования, когда на входе процессора обнаруживаются бесконтекстные синтаксические ошибки[173]. Он запускается при помощи команды **Diagnost/Generate**, когда управляющие граф-схема и таблица прямого просмотра уже построены, т.е. существуют файлы с расширениями .grh, .fp, и, может быть, .mfp и .fcl. При этом, если оптимизированных таблиц прямого просмотра не существует, в диалоговом окне будет запрашиваться, требуется ли генерировать диагностики для неоптимизи-

---

170 В этом случае существует файл с текущим именем и расширением .fp.

171 В этом случае существует также файл с текущим именем и расширением .bp.

172 Которое следует начинать после запуска подсистемы сразу с команды **Minimize**.

173 Контекстные синтаксические ошибки диагностируются другим, но тоже встроенным в процессор, механизмом.

рованного процессора. Если же файлы с расширениями .mfp и .fcl существуют, диагностики будут безоговорочно строиться для оптимизированного процессора. Во время генерации диагностик на экране находится окно статуса с индикацией степени завершенности этой задачи.

Результатом работы такого генератора являются два файла с текущим именем и расширениями .dgn и .pos. Первый из них содержит коллекцию диагностик, а другой — список синтаксических позиций в правилах управляющей грамматики. Этот второй файл необходим при редактировании диагностических сообщений "на фоне" управляющей грамматики.

Редактор диагностических сообщений может быть полезен в том случае, когда желательно использовать нестандартные описания ошибочных ситуаций или несколько вариантов сообщений объединить в одной обобщающей формулировке. Он может быть запущен командой **Diagnost/Edit** после того, как диагностики в стандартной форме сгенерированы. По этой команде открывается окно редактора диагностик, вид которого показан на рис. П.23.

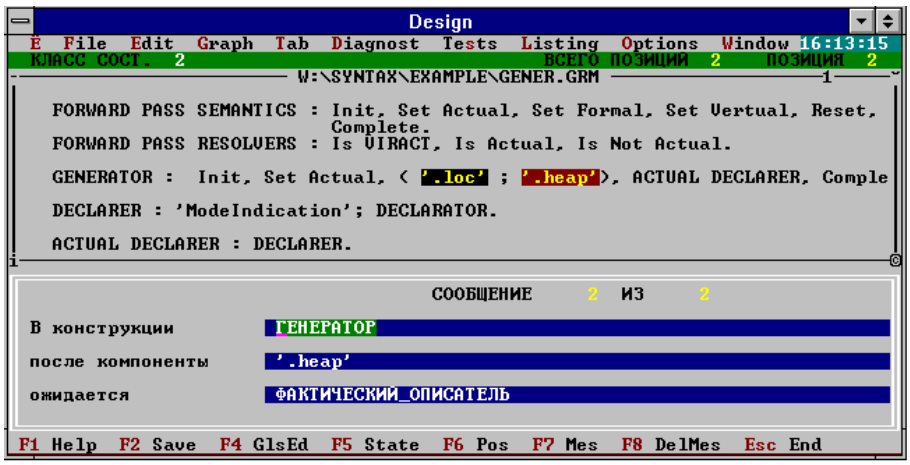


Рис. П.23. Вид окна редактора диагностических сообщений.

Оно состоит из двух частей: верхней, в которой можно просматривать правила управляющей грамматики, и нижней, в которой экспонируются варианты диагностических сообщений. Один класс эквивалентных состояний может включать несколько состояний, а одному состоянию может соответствовать несколько синтаксических позиций (вершин управляющей граф-схемы). Вариант сообщения относится к одной позиции одного состояния одного класса эквивалентности состояний. Результатом редактирования будет новая версия файла с расширением .dgn.

Генератор тестов (прототестов) обеспечивает автоматическую генерацию (прото-) теста заданного уровня для тех нетерминалов грамматики, загруженной в окно редактора, которые отмечены пользователем (рис. П.24).

Рис. П.24. Вид диалогового окна генератора тестов
с отмеченными нетерминалами.

Уровень теста задается в диалоговом окне (рис. П.25), открывающемся по команде **Options / Tests**, которую следует выполнять перед запуском генератора.

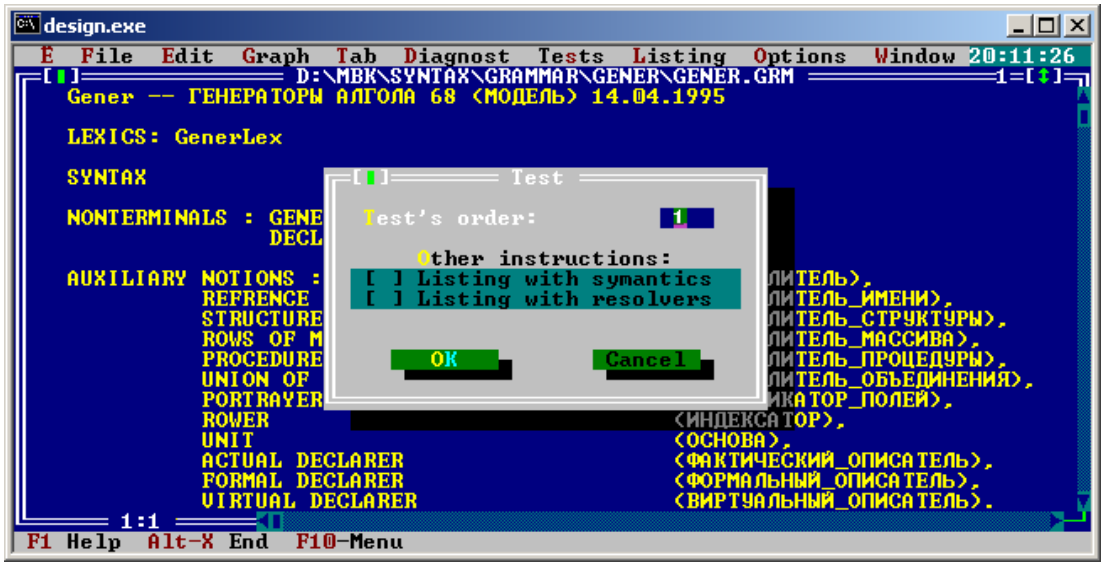


Рис. П.25. Вид диалогового окна задания режима генерации теста.

Генератор листингов производит выдачу документов по проекту одной трансляции, спецификация которой находится в окне редактора. Состав документов, включаемых в листинг, устанавливается пользователем в диалоговом окне, открывающемся по команде **Options / Listing** (рис. П.26). В этом же окне задается размер страницы (в строчках) и направление вывода листинга: в файл (и тогда его можно просматривать по команде **Listing/View**) или непосредственно на принтер.

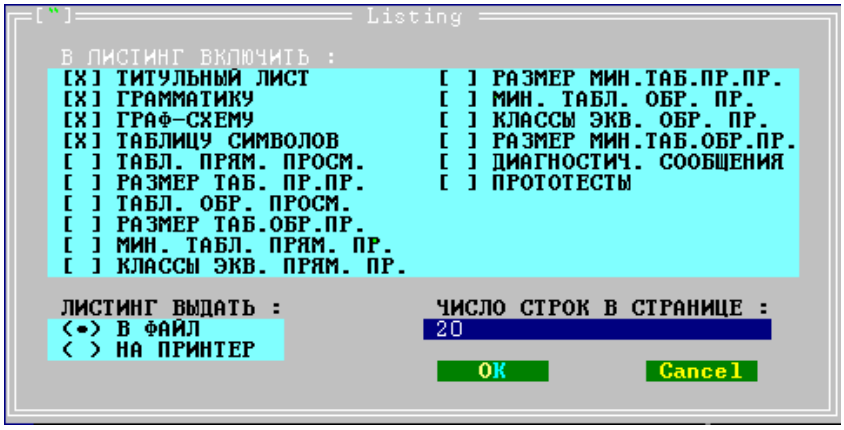


Рис. П.26. Окно настройки содержания листинга.

Общий порядок использования компонент. Как уже отмечалось, подсистема проектирования обеспечивает удобную среду для подготовки синтаксической части спецификации трансляции, ее редактирования и построения управляющих граф-схем, управляющих таблиц, их оптимизации, построения и редактирования диагностических сообщений периода процессирования, генерации тестов, а также получения отчетных документов по всем этим составляющим проекта трансляции.

Стандартный порядок работы в среде проектирования представлен рис. П.27. На схеме двойными прямоугольниками показаны функциональные блоки, двойными стрелками отмечен порядок их использования, одинарными закругленными прямоугольниками обозначены информационные файлы, а одинарными стрелками — потоки данных. Функциональные блоки маркированы именами соответствующих пунктов главного и дополнительных меню подсистемы проектирования, из которых они включаются в работу[174]. Как показывает схема, процесс проектирования трансляции состоит из ряда этапов, выполняемых в определенной последовательности.

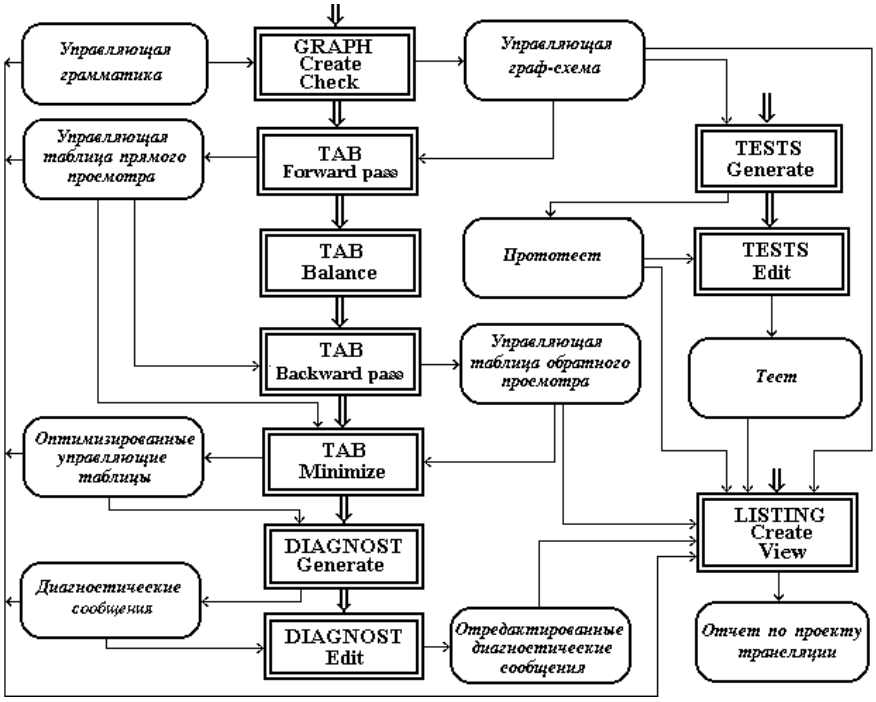
Этап 1: подготовка спецификации трансляции. Прежде всего в окне редактора должна быть подготовлена спецификация трансляции. По крайней мере, должна быть написана управляющая грамматика.

Этап 2: построение управляющей граф-схемы. С помощью функционального блока GRAPH/Create строится управляющая граф-схема. Если функция GRAPH/Create обнаруживает ошибку в управляющей грамматике, то после ее исправления (в окне редактора) попытка построить управляющую граф-схему должна быть повторена.

Этап 3: проверка приведенности управляющей грамматики. После того, как управляющая граф-схема построена, открывается возможность проверить ее приведенность посредством технологической компоненты GRAPH/Check. Если грамматика оказывается неприведенной, то об этом сообщается пользователю, который должен в этом случае исправить управляющую грамматику и повторить предыдущий этап проектирования. Эту проверку следует проводить, если предполагается строить управляющие таблицы процессора.

---

[174] К схеме прилагается краткая расшифровка обозначений.



GRAPH / Create – генератор управляющих граф-схем
GRAPH / Check – проверка приведенности грамматики
TAB / Forward pass – генератор управляющих таблиц прямого просмотра
TAB / Balance – проверка внешней балансировки
TAB / Backward pass – генератор управляющих таблиц обратного просмотра
TAB / Minimize – оптимизатор управляющих таблиц
DIAGNOST / Generate – генератор диагностических сообщений об ошибках
DIAGNOST / Edit – редактор диагностических сообщений об ошибках
TESTS / Generate – генератор (прото-) тестов
TESTS / Edit – редактор тестов
LISTING / Create – генератор отчетов по проектам трансляции
LISTING / View – просмотр отчетов на экране компьютера

Рис. П.27. Общая схема использования подсистемы проектирования технологического комплекса SYNTAX.

Этап 4: построение управляющей таблицы прямого просмотра. После того, как управляющая граф-схема получена, она используется функциональной компонентой TAB / Forward pass для построения управляющей таблицы прямого просмотра. Функция TAB / Forward pass анализирует управляющую граф-схему, и если она не удовлетворяет всем необходимым условиям, выдает диагностическое сообщение о характере нарушений. В таком случае нужно внести требующиеся изменения в управляющую грамматику и снова повторить этапы 2–4.

Этап 5: проверка условия внешней балансировки. Когда управляющая таблица построена, следует проверить, удовлетворяет ли она условию внешней балансировки. Для этого необходимо воспользоваться функциональной компо-

нентой TAB/Balance[175]. Если получено сообщение о нарушении этого условия, нужно изменить управляющую грамматику и повторить все предыдущие этапы снова. Пример такого рода сообщения показан на рис. П.19.

Этап 6: построение управляющей таблицы обратного просмотра. Если в управляющей грамматике используются семантики обратного просмотра или предполагается наблюдать синтаксическую структуру входных цепочек, необходимо воспользоваться функцией TAB/Backward pass для построения управляющей таблицы обратного просмотра. На этом этапе может быть обнаружена семантическая неоднозначность относительно семантик обратного просмотра, и тогда требуется внести необходимые изменения в грамматику и повторить все предыдущие этапы с самого начала, т.е. с построения управляющей граф-схемы.

Этап 7: оптимизация управляющих таблиц. После того, как управляющие таблицы построены, их следует оптимизировать посредством функциональной компоненты TAB/Minimize. Впрочем, от оптимизации можно и отказаться. Например, в том случае, когда хочется иметь более точную диагностику ошибок периода процессирования или предполагается наблюдать синтаксическую структуру входных цепочек.

Если построена только управляющая таблица прямого просмотра, то система запрашивает, действительно ли требуется минимизировать только прямой просмотр, и в случае положительного ответа пользователя производится оптимизация таблицы прямого просмотра. Если же построена и управляющая таблица обратного просмотра, то производится совместная оптимизация управляющих таблиц обоих просмотров. Практика показывает, что оптимизация, как правило, дает значительную экономию памяти.

Заметим, что все формальные ошибки проектирования к моменту выполнения этапа 7 уже устранены. Поэтому на этом и всех последующих этапах никаких ошибок проектирования не диагностируется.

Этап 8: генерация диагностик периода процессирования. Диагностические сообщения генерируются посредством функциональной компоненты DIAGNOST/Generate. Причем к ней можно обратиться сразу после построения управляющих таблиц прямого просмотра или после их оптимизации.

В первом случае сгенерируются диагностики, рассчитанные на использование при процессировании под управлением неоптимизированных таблиц.

Во втором — они годятся лишь при процессировании под управлением оптимизированных таблиц. Рекомендуется строить диагностики в двух вариантах, чтобы во время отладки можно было оперативно переключаться с одних таблиц на другие.

Например, если захочется посмотреть на синтаксическую структуру входной цепочки, придется использовать прямой и обратный просмотр с неоптимизированными управляющими таблицами.

---

[175] Если грамматика явнорегулярна, такая проверка излишня.

Этап 9: редактирование диагностик периода процессирования. Если желательно отредактировать автоматически сгенерированные сообщения об ошибках, то следует воспользоваться технологической функцией DIAGNOST/Edit. Это может потребоваться, если, например, при генерации диагностик использовался шаблон вида “В ... после ... ожидается ...” , требующий подгонки окончаний диагностических терминов для согласования их с установленным шаблоном.

Этап 10: генерация оптимального прототеста возможна сразу после построения управляющей граф-схемы при помощи функции TESTS/Generate[176].

Этап 11: редактирование прототеста. Для построения исполнимого теста, исходя из полученного прототеста, можно воспользоваться технологической функцией TESTS/Edit[177].

Этап 12: генерация отчета по проекту трансляции. В любой момент можно получить отчет по готовой части проекта трансляции, если воспользоваться функцией LISTING/Create. Его можно просмотреть на экране компьютера при помощи функции LISTING/View, если листинг был направлен в файл. По выбору пользователя он может быть также выдан на печать.

3. ПОДСИСТЕМА ПРОЦЕССИРОВАНИЯ

Подсистема процессирования включает средства для подготовки и редактирования спецификации микролексики, описания операционной среды, генерации микролексических классов для штатного транслитератора и компиляции операционной среды, а также для компоновки действующих SYNTAX-приложений из составляющих их компонент. К ним относятся транслитераторы, параметризуемые микролексическими классами, и управляющие процессоры (конечные или сплайновые, простые или челночные, анализирующие или порождающие), параметризуемые управляющими таблицами, диагностическими сообщениями и операционными средами. Подсистема процессирования обеспечивает также благоприятную обстановку для наблюдения за работой готовых средств СУОД в процессе их тестирования, предоставляя возможность протоколирования и потактового исполнения на любых уровнях процессирования. Подсистема запускается командой **Process** [*Name.grm*], где *Name.grm* — имя файла некоторой TSL-спецификации. По этой команде появляется заставка, вид которой показан на рис. П.28.

---

176 Однако надо иметь в виду, что при этом контекстные условия не учитываются, даже если в спецификации трансляции используются резольверные символы. Учет контекста при генерации тестов — дело дальнейших исследований.

177 Пока эта функция не реализована.

Рис. П.28. Вид заставки подсистемы процессирования.

После сброса заставки клавишей Esc, открывается панель этой подсистемы, включающая три основных элемента управления: полосу меню, собственно поверхность панели и строку статуса (рис. П.29).

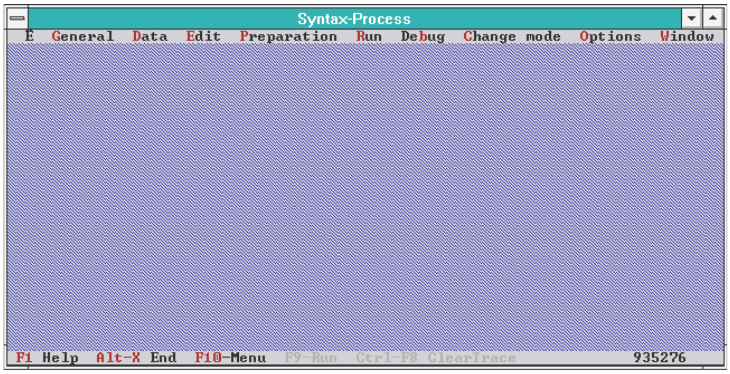


Рис. П.29. Вид панели подсистемы процессирования.

Если в команде **Process** был указан параметр, то в эту панель вставляется окно редактора, содержащего указанную спецификацию (рис. П.30).

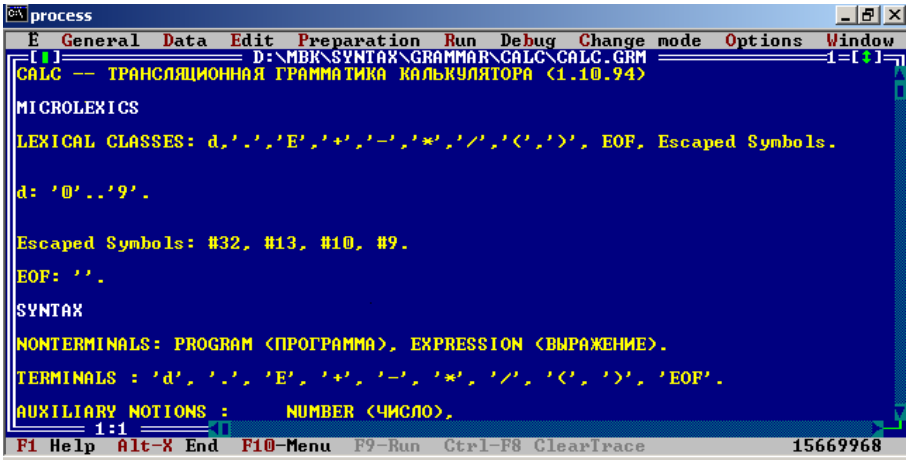


Рис. П.30. Вид панели подсистемы процессирования
с TSL-спецификацией калькулятора.

Если TSL-спецификация не была загружена при запуске подсистемы процессирования, т.е. в команде **Process** не был задан параметр, то это необходимо сделать при помощи команды **General/Load processor**, так как в любом случае работа над конкретной TSL-спецификацией возможна лишь после ее загрузки в окно редактора подсистемы. Одновременно с этим подсистеме процессирования открывается доступ к соответствующей управляющей граф-схеме.

Система меню и режимы работы. Здесь мы приведем полный перечень всех команд среды подсистемы процессирования со спецификацией их назначения.

Подсистема процессирования может находиться в одном из двух функциональных режимов:

1) режиме подготовки процессоров к работе,
2) режиме отладки.

Режимы работы отличаются друг от друга перечнем допустимых команд. Но некоторые команды, например команды из разделов меню About (≅), Edit, Options Edit и Options Window, доступны для использования в обоих режимах.

В режиме подготовки доступны команды из разделов General и Preparation, позволяющие провести полный цикл подготовки к работе процессора или комплекса процессоров, составляющих данное SYNTAX-приложение. Перечень операций этого цикла описан далее.

В режиме отладки открывается доступ к командам из разделов Data, Run и Debug основного меню, которые инициируют запуск подготовленного SYNTAX-приложения и предоставляют средства отладки, имеющиеся в интегрированной среде подсистемы процессирования.

Раздел About (Ё) — чисто информационный. Он содержит команду **About**, выбор которой влечет появление диалогового окна с информацией о версии подсистемы процессирования (см. рис. П.31).

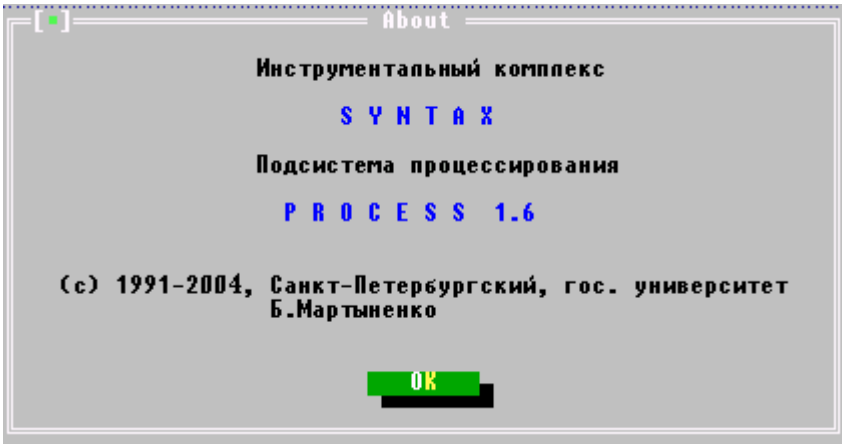


Рис. П.31. Информация о версии подсистемы процессирования.

Раздел General. Команды раздела General (рис. П.32) предназначены для загрузки спецификации трансляции, ее сохранения после редактирования, а

также временной приостановки работы с подсистемой процессирования (временного выхода в DOS) и завершения работы с ней. Рассмотрим эти команды подробнее.

Команда **Load processor** (F3[178]) влечет появление стандартного диалогового окна со списком TSL-спецификаций. Пользователю предлагается выбрать один из существующих в текущем (или любом другом) каталоге файлов с расширением .grm (рис. П.33). Выбранная спецификация загружается в окно текстового редактора. Подобным образом можно открыть несколько окон одновременно. При этом команды из разделов меню Edit и Preparation будут относиться к активному (т.е. выделенному двойной рамкой) окну.

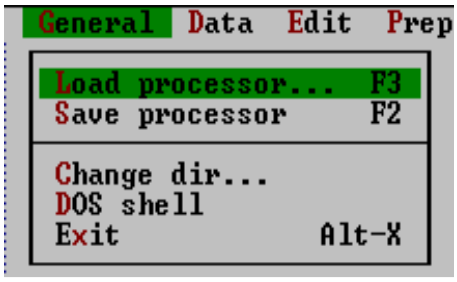


Рис. П.32. Меню General.

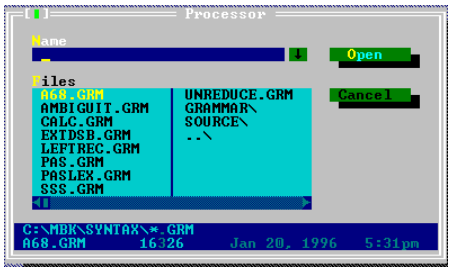


Рис. П.33. Диалоговое окно выбора спецификации трансляции.

Рекомендуется работать с текущем каталогом, в котором размещены как файлы, созданные подсистемой проектирования, так и файлы, вновь создаваемые в подсистеме процессирования.

Команда **Save processor** (F2) сохраняет текст спецификации трансляции из текущего окна редактора в соответствующем файле на диске.

Команда **Change Dir** служит для перехода в другой каталог. После выполнения этого пункта меню появляется диалоговое окно (рис.П.34), в котором можно выбрать любой из существующих каталогов. Затем во вновь установленном каталоге можно выбрать нужный файл TSL-спецификации.

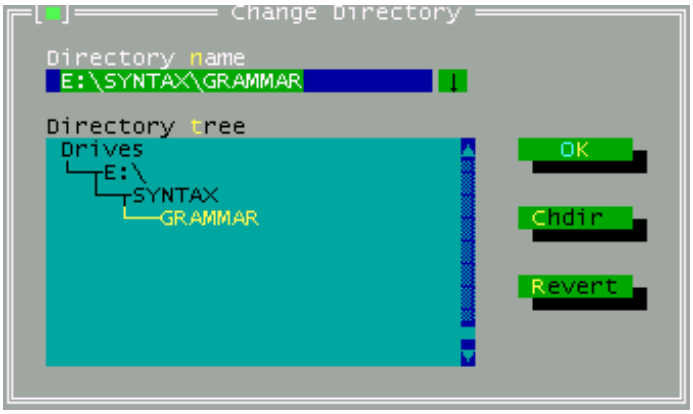


Рис. П.34. Общий вид диалогового окна установки нового каталога.

178 В скобках указываются "горячие" клавиши, нажатие которых инициирует исполнение соответствующих команд.

Команда **DOS Shell** позволяет осуществить временный выход в среду DOS. Возврат в среду подсистемы процессирования производится по команде DOS **Exit**.

Команда **Exit** предназначена для завершения сеанса работы в подсистеме процессирования. При наличии не сохраненных данных в окнах редактора подсистема выдает соответствующее предупреждение с предложением сохранить эти данные (рис. П.35).

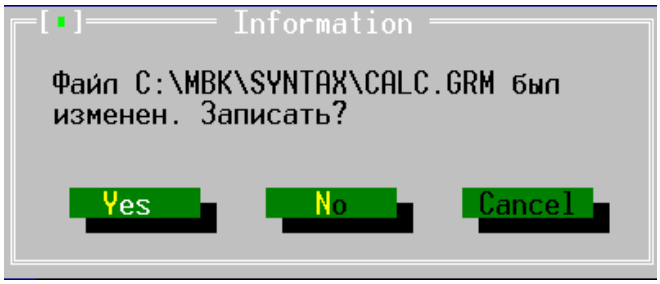


Рис. П.35. Предупреждение о существовании не сохраненных данных.

Раздел Edit. Команды этого раздела служат для редактирования содержимого текущего активного окна. Действие команд одинаково для окна со спецификацией трансляции и окна исходных данных, содержащего входную цепочку. Команды доступны, если на панели экрана открыто хотя бы одно окно текстового редактора. Для улучшения восприятия структуры текста TSL-спецификации редактор выделяет ключевые слова, которыми начинаются ее разделы, особым цветом.

Команда **Undo** (Alt + Back Space) влечет отмену сделанных изменений в тексте с момента последнего перемещения курсора. Если таковых изменений не было, команда недоступна.

Команда **Cut** (Shift + Del) производит перенос выделенного текста во временный буфер редактора (Clipboard). Эта команда доступна, если в тексте имеется выделенный участок. Буфер редактора — один на все окна редактора и существует в течение всего сеанса работы подсистемы, однако его содержимое не сохраняется между сеансами работы. Перенесенный во внутренний буфер обмена текст остается там выделенным до нового перемещения, копирования или изменения в окне буфера обмена (см. команду **Show clipboard**). Содержимое временного буфера может быть вставлено в нужное место редактируемого текста при помощи команды **Paste**.

Команда **Copy** (Ctrl + Ins) производит копирование выделенного текста во временный буфер редактора (Clipboard). В остальном команда аналогична команде **Cut**.

Команда **Paste** (**Shift + Ins**) вставляет текст из внутреннего буфера редактора в окно редактирования. Вставка начинается с текущей позиции курсора.

Команда **Clear** (Ctrl + Del) удаляет выделенный участок текста без перемещения его во внутренний буфер обмена.

Команда **Show clipboard** открывает окно внутреннего буфера обмена.

Команда **Search** (Alt + S) влечет появление диалогового окна ввода образца для поиска текста в активном окне редактора. Окно содержит также переключатели для настройки параметров поиска.

Команда **Replace** (Alt + R) требует задания образца для поиска (Text to find) и образца для замены (New text) в открываемом ею диалоговом окне (рис. П.36). В нем можно установить режимы исполнения этой команды:

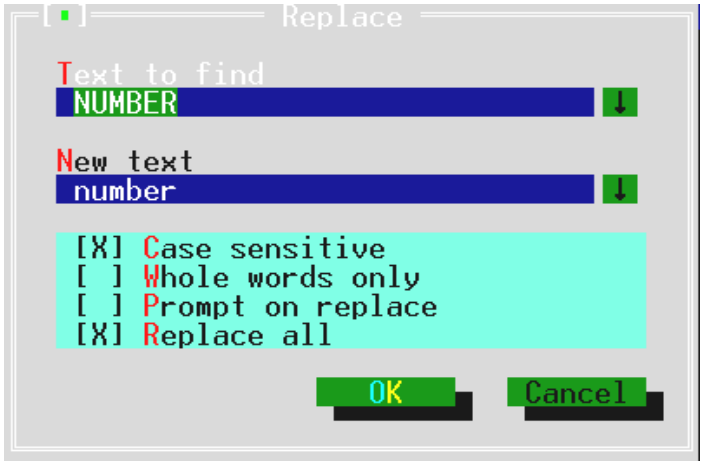


Рис.П.36. Окно команды **Edit / Replace**.

Case sensitive — поиск образца с учетом / без учета регистра.

Whole words only — поиск целого слова или только его части.

Prompt on replace — запрашивать разрешение или нет для каждого вхождения образца замены.

Replace all — замещать все вхождения образца поиска или нет.

Команда **Search again** (Alt + L) позволяет повторить операцию, которая была инициирована одной из двух предыдущих команд. Таким образом можно найти следующее вхождение образца в тексте или осуществить очередную замену.

Кроме команд, собранных в разделе Edit основного меню, окно редактора поддерживает следующие команды быстрого набора:

Команды, управляющие перемещением курсора:

Вправо	**Ctrl + D**
Влево	**Ctrl + S**
Вверх	**Ctrl + E**
Вниз	**Ctrl + X**
Вправо на слово	**Ctrl + F**, **Ctrl + □**
Влево на слово	**Ctrl + A**, **Ctrl + □**
В начало строки	**Ctrl + Q S**, **Home**
В конец строки	**Ctrl + Q D**, **End**

Команды листания текста:

В начало файла	**Ctrl + Q R**, **Ctrl + PgUp**
В конец файла	**Ctrl + Q C**, **Ctrl + PgDn**
На экран вверх	**Ctrl + R**, **PgUp**
На экран вниз	**Ctrl + C**, **PgDn**

Команды управления удалением/вставкой:

Удалить символ слева от курсора	**Ctrl+H**, **Back Space**
Удалить символ в позиции курсора	**Ctrl+G**, **Del**
Удалить слово справа от курсора	**Ctrl+T**
Удалить строку	**Ctrl+Y**
Удалить часть строки перед курсором	**Ctrl+Q H**
Удалить часть строки после курсора	**Ctrl+Q Y**
Переключение режима замены/вставки	**Ctrl+V**, **Ins**

Команды работы с блоками:

Отметить начало блока	**Ctrl+K B**
Отметить конец блока	**Ctrl+K K**, **Ctrl+Ins**
Включить/выключить подсветку блока	**Ctrl+K H**
Вставить блок	**Ctrl+K C**, **Shift+Ins**
Удалить блок	**Ctrl+K Y**, **Shift+Del**

Специальные команды:

Разбить строку в позиции курсора	**Ctrl+M**, **Enter**
Переключение режима Indent Mode	**Ctrl+O**

Раздел Preparation содержит команды для подготовки к работе процессоров, входящих в состав одного SYNTAX-приложения (рис. П.37).

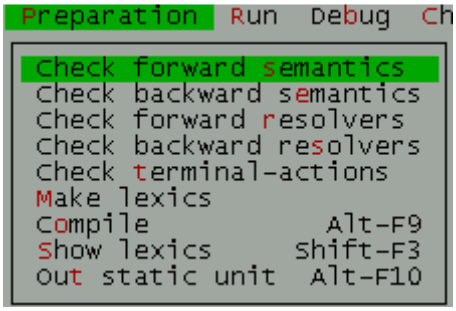


Рис. П.37. Меню Preparation.

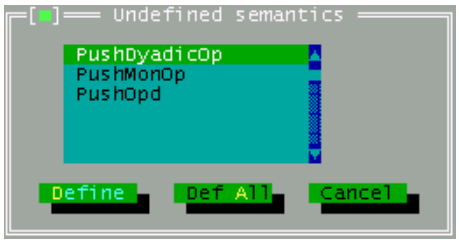


Рис. П.38. Окно не описанных семантик.

Команда **Check (Forward/Backward) semantics** выполняет сканирование текста спецификации трансляции с целью выявления семантических символов, используемых в управляющей грамматике, реализация которых не определена. В случае обнаружения таких семантических символов, все они выводятся на экран с предложением определить их реализацию (рис. П.38). В этот момент можно отметить один из них и нажать кнопку Define или сразу нажать кнопку Def All. В результате, начиная с текущей позиции курсора, в спецификацию будут вставлены один или несколько шаблонов вида

procedure Sem; **begin** ... **end**;

где Sem — один из семантических символов, указанных в окне Undefined semantics. Пользователю остается лишь заполнить эти шаблоны надлежащим образом.

Команда **Check (Forward/Backward) resolvers** полностью аналогична предыдущей, но относится к резольверным символам. Вставляемые в текст шаблоны для резольверов, реализация которых не определена, имеют вид

function Res : **boolean**; **begin** Res := **true end**;

где Res — один из резольверных символов, указанных в окне Undefined resolvers.

Команда **Check terminal-actions** актуальна для порождающих грамматик и аналогична **команде Check Forward semantics**.

Команда **Make lexics** выполняет поиск раздела описания лексики в текущей спецификации трансляции. Если этот раздел начинается с ключевого слова **MICROLEXICS**, то запускается встроенный анализатор микролексики, который проверяет правильность спецификации и создает файл микролексики для данного процессора. Если в описании микролексики обнаруживается ошибка, то в красной строке, расположенной в верхней части экрана, выдается предупреждающее сообщение, а место ошибки указывается при помощи выделения не принятого символа (рис. П.39).

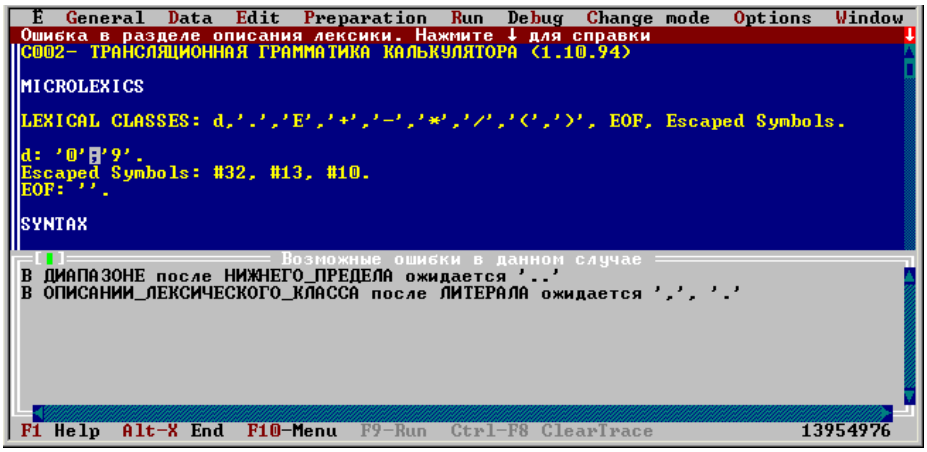


Рис. П.39. Диагностика ошибки в описании микролексики.

При желании уточнить характер ошибки можно нажать клавишу ‘↓’, и тогда открывается окно ‘*Возможные ошибки в данном случае*’ с вариантами диагностических сообщений о найденной ошибке.

Если раздел спецификации лексики начинается с ключевого слова **LEXICS**, то производится связывание текущего процессора с тем, имя которого указано после этого ключевого слова. В случае успешного завершения команды **Make lexics** выдается информационное сообщение о подсоединении к данному процессору соответствующего лексического анализатора.

Команда **Compile**[179] (Alt+F9) выполняет анализ разделов Environment, Implementation , и Messages, исходного текста спецификации трансляции с целью создания библиотечного модуля с описанием операционной среды данного процессора. Затем это описание компилируется в объектный DLL-модуль компилятором BPC фирмы Borland. Все сообщения от него перехватываются подсистемой процессирования. В случае обнаружения синтаксической ошибки ее место указывается курсором в исходном тексте спецификации, находящемся в окне редактора подсистемы процессирования (рис. П.40).

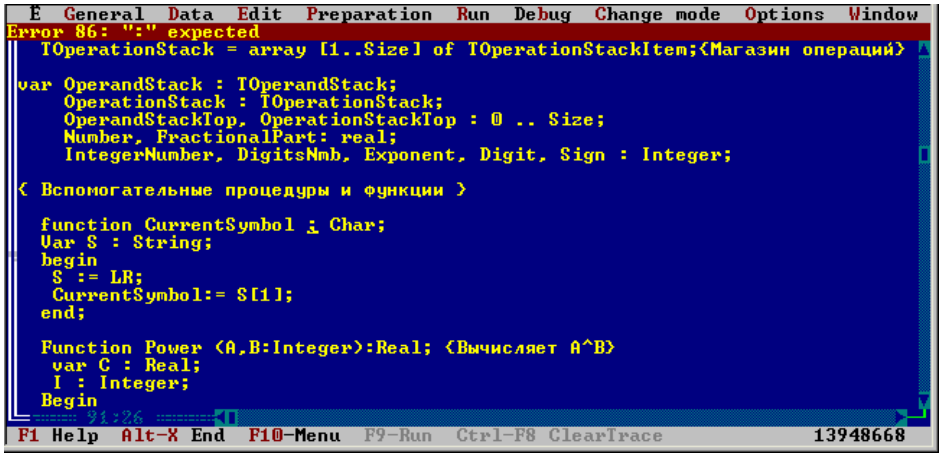


Рис. П.40. Сообщение об ошибке периода компиляции.

При этом сообщение компилятора об ошибке появляется в красной строке в верхней части окна редактора. Если позиционирование места ошибки невозможно, сообщение выводится в отдельном окне.

Особенности настройки среды подсистемы процессирования для вызова компилятора BPC приведены далее.

Команда **Show lexics** (Shift+F3) позволяет просмотреть лексические входы любого процессора, для которого уже построена управляющая граф-схема. Выбор этой команды влечет появление стандартного диалогового окна для выбора файла управляющей граф-схемы. Лексические (входные) классы выбранного процессора отображаются в виде списка в отдельном окне ‘Лексика процессора’ (рис. П.41). Эта команда введена для упрощения согласования лексических входов/выходов двух разных процессоров, если один из них проектируется в качестве сканера для другого.

Команда **Change mode**. Как отмечалось ранее, подсистема процессирования может находиться в одном из двух режимов: подготовительном и отладочном.

Переход из режима подготовки в режим отладки (и наоборот) осуществляется выбором команды **Change mode** (Alt+Space) основного меню.

Для перехода в режим отладки необходимо, чтобы имя главного процессора было определено. Если в режиме подготовки было открыто одно или несколько

[179] По команде **Out static unit** компилируется статический модуль (.tpp), а не модуль динамической линковки .dll, как по команде Compile.

окон с TSL-спецификациями, то главным считается процессор, реализующий трансляцию, специфицированную в активном окне.

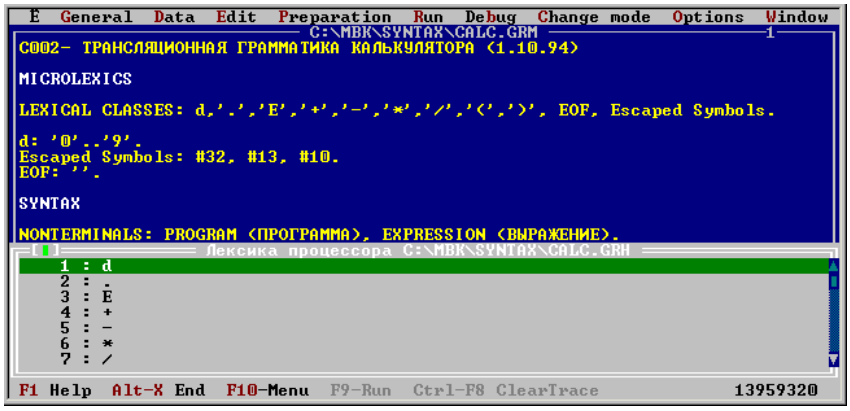


Рис. П.41. Вид окна с лексикой процессора CALC.

Если в режиме подготовки не было открыто ни одного окна с TSL-спецификацией, то при выполнении команды **Change mode** будет предложено указать имя главного процессора в стандартном диалоговом окне выбора имени файла. В случае отказа от выбора среда останется в режиме подготовки процессоров к работе.

В режиме отладки на панели автоматически открываются три окна: окно входного потока (Input), окно выходного потока (Output) и окно трассировочных сообщений (Trace). Стандартный вид экрана в режиме отладки изображен на рис. П.42.

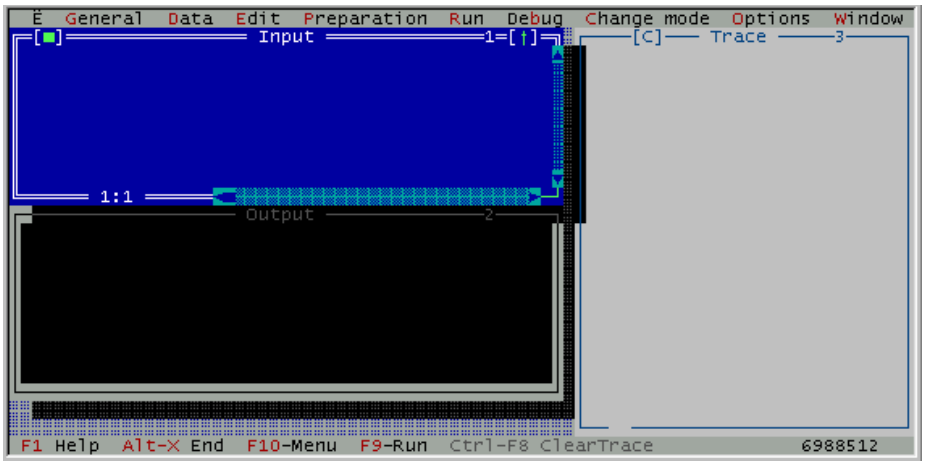


Рис. П.42. Стандартный вид панели подсистемы процессирования в режиме отладки.

Если ранее окну Input было назначено имя файла, то при новом входе в режим отладки содержимое этого файла будет отображено в окне входного потока.

При возврате в режим подготовки автоматически загружается исходный текст спецификации трансляции, реализуемой главным процессором.

При смене режима все окна, открытые в предыдущем режиме, закрываются.

Раздел Data. Команды этого раздела (рис.П.43) доступны только в режиме отладки и служат для манипуляции с входными и выходными данными.

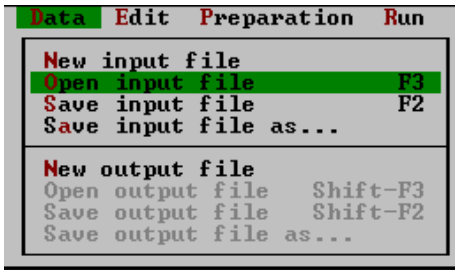


Рис. П.43. Меню Data.

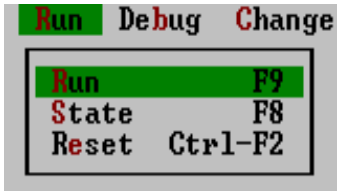


Рис. П.44. Меню Run.

Команда **New input data** открывает новое окно входных данных Input, которое является окном текстового редактора. Одновременно может быть открыто только одно окно входных данных. Поэтому если во время открытия нового окна входных данных такое уже существует, то оно (уже существующее) будет закрыто. Команда **Open input file** (F3) позволяет открыть новое окно входных данных и загрузить туда текст из файла, указанного пользователем в стандартном диалоговом окне. В остальном эта команда аналогична предыдущей.

Команда **Save input data** сохраняет данные из окна входных данных в файле на диске. Если окну еще не присвоено имя, то будет выполнена команда **Save input data as** (см. ниже).

Команда **Save input data as** сохраняет данные, находящиеся в окне исходных данных, в файле на диске, указанном пользователем в стандартном диалоговом окне.

Команда **New output file** создает новое окно выходных данных Output. Одновременно может быть открыто только одно окно выходных данных. Поэтому в случае, если такое окно уже было создано ранее, оно будет закрыто. Используя эту команду, можно очистить окно вывода Output.

Команда **Save output file** позволяет сохранить результат работы SYNTAX-приложения, находящийся в окне Output, в текстовом файле на диске. Если окну выходных данных еще не было присвоено имя, то будет выполнена команда **Save output file as**.

Команда **Save output file as** сохраняет данные из окна результата Output в текстовом файле на диске, указанном пользователем в стандартном диалоговом окне.

Раздел Run основного меню (рис. П.44) содержит три команды: **Run**, **State** и **Reset**.

Команда **Run** (F9) инициализирует все процессоры, входящие в состав данного SYNTAX-приложения и производит запуск основного процессора. *Основным процессором* SYNTAX-приложения называется процессор, который не используется никаким другим процессором в качестве сканера или семантики. Его уровень равен 1. Уровни подчиненных ему процессоров нумеруются значениями 2, 3, ... и т.д.

Во время инициализации проверяется наличие всех необходимых файлов, содержащих управляющие таблицы, диагностические сообщения, операционные среды и т.д., с учетом последовательности их создания (по дате и времени). Устаревшие компоненты считаются отсутствующими. Недостающие компоненты, если возможно, автоматически восстанавливаются. Например, если устарел файл лексики, а описание лексики в исходной спецификации существует, то делается попытка обновить этот файл.

Выполнение производится либо до достижения главным процессором конечного состояния, либо до возникновения ошибочной ситуации в каком-то из процессоров. В последнем случае выдается диагностическое сообщение соответствующего уровня с указанием места ошибки при помощи курсора в окне исходных данных (рис. П.45).

Если трансляция специфицируется порождающей грамматикой, то автоматически запускается порождающий процессор. При этом рекомендуется отключать генерацию протокола (Options/Debagger/Trace) и выдачу его в листинг (Options/Run), чтобы избежать переполнения представляющих их массивов.

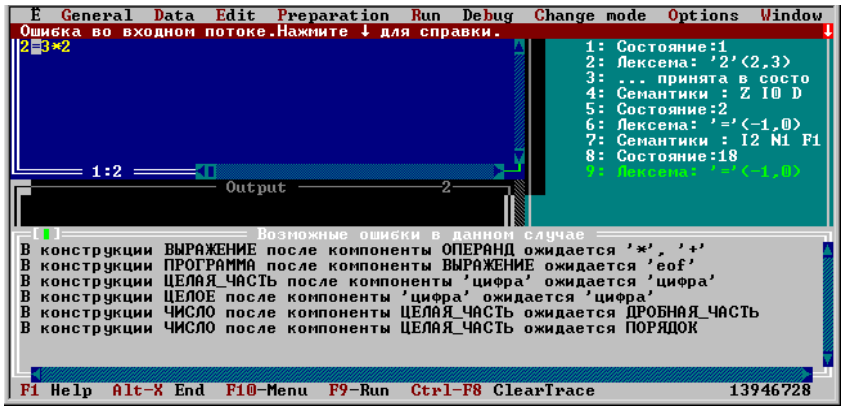


Рис. П.45. Диагностика бесконтекстной ошибки периода процессирования.

Команда **State** (**F8**) аналогична предыдущей, однако работа главного процессора происходит пошагово, с остановкой в каждом состоянии управления. Если в текущем состоянии запрашивается очередная лексема, поставляемая подпроцессором-сканером, или какая-нибудь семантика реализуется процессором, то эти вспомогательные процессоры отрабатывают не пошагово. Все шаги процессирования протоколируются в окне трассировочных сообщений в соответствии с заданным режимом трассировки (см. далее) для каждого процессора, входящего в состав данного SYNTAX-приложения.

Команда **Reset** (Ctrl + F2) приостанавливает пошаговое исполнение трансляции и запрашивает пользователя, остановить ли процесс (рис. П.46). Пользователь может согласиться — и тогда главный процессор возвращается в начальное состояние, или не согласиться — и тогда трансляция может быть продолжена как в пошаговом (F8), так и в автоматическом (F9) режиме.

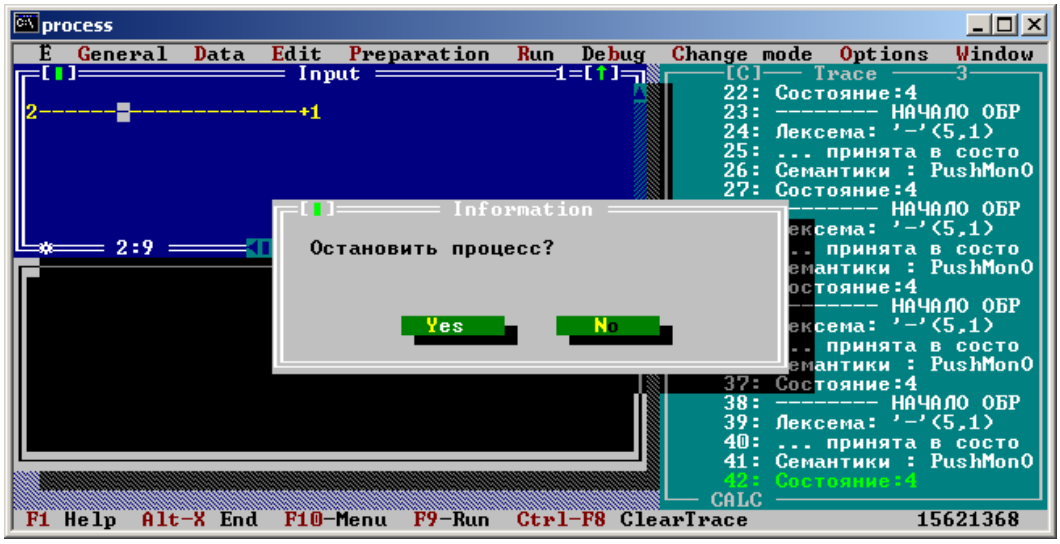


Рис. П.46. Окно команды **Run/Reset**

Раздел Debug основного меню содержит команды управления окном трассировочных сообщений (рис. П.47). Они доступны только в том случае, если окно Trace распахнуто на экране и в нем имеется хотя бы одно сообщение.

Команда **Save trace info** позволяет сохранить содержимое окна трассировочных сообщений в файле с указанным именем.

Команда **Clear trace window** (Ctrl+F8) очищает окно трассировочных сообщений.

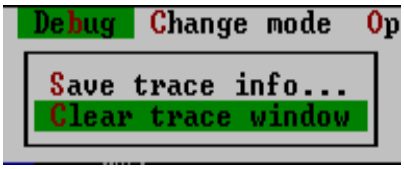


Рис. П.47. Меню Debug.

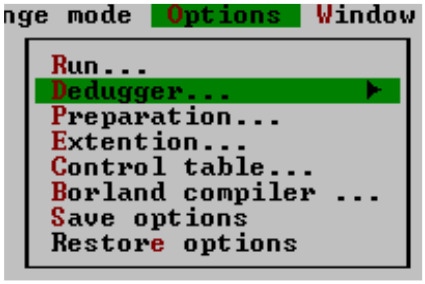


Рис. П.48. Меню Options.

Раздел Options. Здесь собраны команды настройки интегрированной среды подсистемы процессирования (рис. П.48). Все текущие установки могут быть сохранены между сеансами работы.

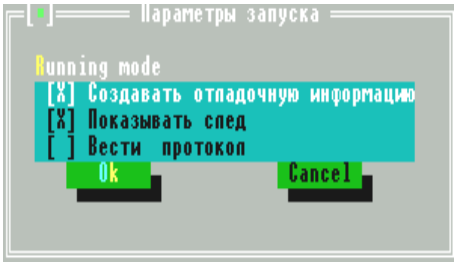


Рис. П.49. Окно команды **Options/Run**.

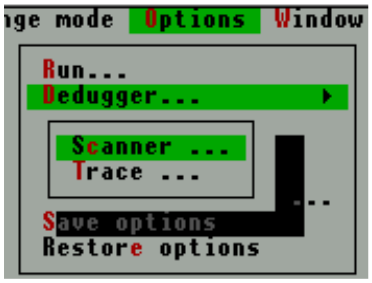


Рис. П.50. Окно команды **Options/Debugger**.

Команда **Run** влечет появление диалогового окна с набором возможных режимов запуска (рис. П.49). В этом диалоговом окне можно указать, требуется ли генерировать отладочную информацию, следует ли показывать “бегунок” — маркер, отмечающий позицию сканирования входного текста, и нужно ли создавать файл протокола[180].

Команда **Debugger** определяет настройки встроенного отладчика. Он содержит два подпункта (рис. П.50).

Подпункт Scanner позволяет настроить работу встроенного транслитератора подсистемы процессирования (рис. П.51). Именно, в дополнение к списку символов, определенному в разделе описания микролексики как символы, относящиеся к классу Escaped Symbols, которые игнорируются транслитератором, в открывающемся окне можно задать еще один список такого же рода символов и указать, каким образом следует использовать эти два списка:

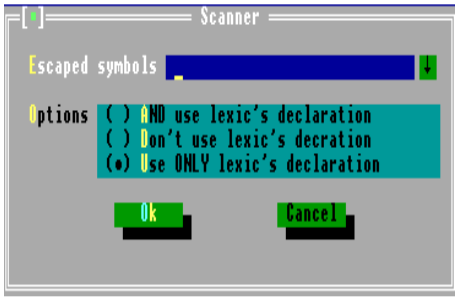


Рис. П.51. Диалоговое окно настройки транслитератора.

Рис. П.52. Диалоговое окно настройки выдачи трассировочной информации.

- использовать только список, заданный описанием класса Escaped Symbols в разделе MICROLEXICS TSL-спецификации (если такого класса нет, то список игнорируемых литер пуст);
- используются оба списка;
- используется только список, определенный в окне настройки Scanner.

Подпункт Trace определяет состав трассировочной информации для процессоров различных уровней (рис. П.52). При этом полагается, что главный процессор имеет уровень 1, его сканер — 2 и т.д. Для каждого из уровней можно включить или выключить отображение следующей информации: номер текущего состояния; сканируемую лексему и момент ее приема текущим процессором; выполняемые семантики; тестируемые резольверы; пересылаемые между процессорами сообщения.

Отметим, что перечень отображаемых в трассировочном окне сообщений можно изменять уже в процессе исполнения SYNTAX-приложения.

Команда **Extention** определяет расширения имен файлов, входящих в SYNTAX-приложения (рис. П.53). По умолчанию они такие же, как в подсистеме проектирования (Designing subsystem**)**.

---

180 Он имеет имя соответствующей спецификации с расширением .prt.

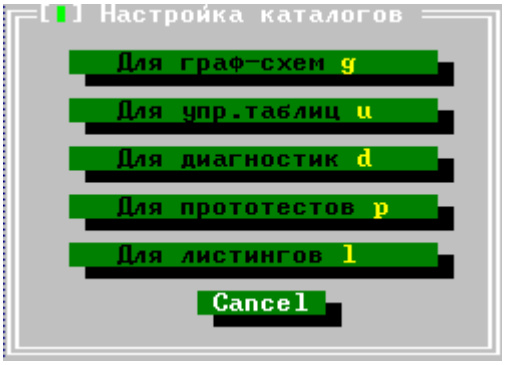


Рис. П.53. Установка расширений имен файлов.

Команда **Control tables** в открывающемся окне (рис. П.54) показывает список процессоров[181], для каждого из которых можно

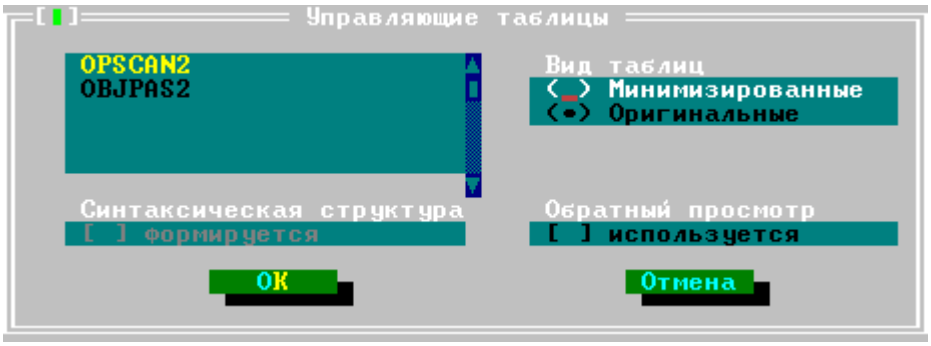


Рис. П.54. Окно установки управляющих таблиц.

задать вид управляющих таблиц (оригинальные / минимизированные), которые будут им использоваться при работе, а также заказать построение синтаксической структуры входных цепочек в терминах соответствующей грамматики[182].

Команда **Save options** используется для сохранения настроек. Она создает и помещает файл конфигурации process.cfg в текущий рабочий каталог.

Команда **Restore options** производит поиск файла конфигурации process.cfg в текущем рабочем каталоге, и, в случае успеха, восстанавливает оттуда настройки.

Команда **Borland compiler** открывает диалоговое окно (рис. П.55), в котором можно задать полный путь к компилятору bpc и режимы его работы. Обычно в этом окне уже имеются некоторые установки. Если необходимо, они могут быть изменены.

---

[181] Черта отделяет имена главного процессора и подчиненных ему процессоров от других процессоров, спецификации которых имеются в текущем каталоге. Установки относятся к процессору, имя которого выбрано в списке.

[182] В этом случае по умолчанию устанавливается использование оригинальных управляющих таблиц прямого и обратного просмотров.

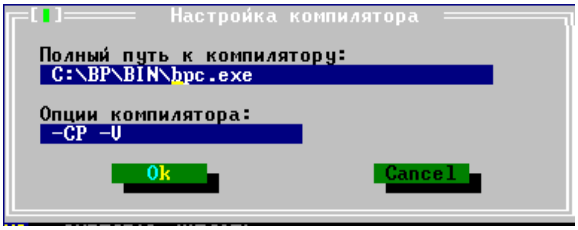


Рис. П.55. Окно настройки компилятора.

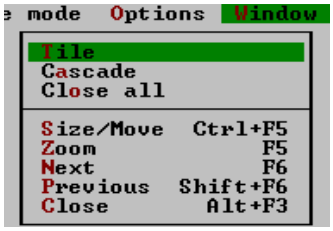


Рис. П.56. Меню Window.

В строке ввода пути должен быть установлен полный путь до компилятора bpc, например, C:\BP\BIN\bpc.exe. В строке ввода опций компилятора можно задать желаемые режимы, например: –CP –V.

Поиск компилятора производится сначала в каталогах, перечисленных в переменной окружения DOS PATH. Если он безуспешен, то берется путь из вышеупомянутого поля ввода. Если путь не задан или задан неверно, выдается сообщение об ошибке.

Раздел Window содержит традиционные команды для управления окнами на экране (рис. П.56).

Команда **Tile** размещает окна таким образом, чтобы все они одновременно уместились на панели экрана.

Команда **Cascade** размещает окна одно над другим так, что верхняя часть рамки, содержащая имя файла и другую служебную информацию, остается незакрытой другими окнами.

Команда **Close all** закрывает все окна панели экрана.

Команда **Size/Move** (Ctrl+F5) позволяет изменять размеры и месторасположение текущего активного окна.

Команда **Zoom** (F5) позволяет распахнуть текущее активное окно до размеров панели экрана.

Команда **Next** (F6) выбирает следующее по порядку окно, т.е. делает его активным.

Команда **Previous** (Shift+F6) делает активным предыдущее окно.

Команда **Close** (Alt+F3) закрывает текущее активное окно.

Общий порядок работы. После того, как работа над синтаксической частью спецификации трансляции завершена, необходимо перейти в подсистему процессирования для того, чтобы в режиме подготовки закончить проектирование трансляции, а в режиме отладки посмотреть на построенное SYNTAX-приложение в действии.

Стандартный порядок работы пользователя в подсистеме процессирования состоит из следующих основных этапов.

Этап 1: спецификация лексики (микролексики) . Прежде всего, в окно редактора подсистемы процессирования, находящейся в режиме подготовки, необходимо загрузить управляющую грамматику и пополнить ее описанием лексики или микролексики в зависимости от того, в какой поддержке нуждается процессор, определяемый данной управляющей грамматикой. После заголовка грамматики следует добавить описание микролексики, если ему

требуется транслитератор, или ссылку на спецификацию сканера, если используется сканер[183].

Этап 2: спецификация операционной среды. Необходимо описать операционную среду, интерпретацию контекстных символов и обработчики сообщений, если таковые используются в данной управляющей грамматике[184]. Технологическая компонента Preparation позволяет проверить, для всех ли контекстных символов определена интерпретация. Если нет — в диалоговом окне выдается список не определенных контекстных символов[185] (см., например, рис. П.38). В этом же окне предлагается автоматическая генерация шаблонов недостающих описаний в виде процедур с пустыми телами — для семантик, и логических функций (с результатом **true**) — для интерпретации резольверных символов. По выбору пользователя такие шаблоны вставляются в спецификацию трансляции на место текущего положения курсора.

При написании семантик сканеров, формирующих выходные лексемы, необходимо знать номера соответствующих лексических классов. Команда **Preparation / Show lexics** позволяет просмотреть полный список лексических классов по всем грамматикам, по которым построены граф-схемы.

Этап 3: компиляция операционной среды — завершает подготовку одной компоненты SYNTAX-приложения.

Выполнив этапы 1–3 для каждой компоненты SYNTAX-приложения, можно переходить в режим процессирования для наблюдения за его работой. Для этого нужно проследить за тем, чтобы в окне редактора находилась спецификация трансляции, которая считается главной[186], и затем посредством команды **Change mode** перевести подсистему процессирования в режим отладки.

Этап 4: отладка SYNTAX-приложения. Установив желательные опции процессирования (разд. Option), и задав входную цепочку в окне ввода, можно запустить автоматическое (по команде **Run / Run**) или пошаговое (по команде **Run / State**) исполнение загруженного SYNTAX-приложения или некоторой его компоненты. При этом в окне трассировки можно наблюдать ход обработки данной входной цепочки.

Все этапы работы в подсистеме процессирования контролируются, и в случае обнаружения каких-либо ошибок, выдаются соответствующие диагностические сообщения.

---

[183] Фактически описание лексики или микролексики могло быть включено в спецификацию трансляции еще при работе в подсистеме проектирования, но его использование по существу происходит только при исполнении команды **Preparation / Make lexics** в подсистеме процессирования.

[184] Это описание могло быть сделано и при работе в подсистеме проектирования, но по существу оно используется лишь при генерации библиотечного модуля, описывающего операционную среду данного процессора.

[185] Эта проверка выполняется посредством команд **Preparation /Check semantics** и **Preparation /Check resolvers**.

[186] Главной трансляцией полезно считать поочередно все трансляции, начиная с самого нижнего уровня, и заканчивая самым высоким. Другими словами, рекомендуется отлаживать сначала процессоры, у которых нет подчиненных, затем те, которые используют уже отлаженные процессоры.

4. ИЛЛЮСТАЦИЯ ПАРАДИГМЫ ОБЪЕКТНО-СИНТАКСИЧЕСКОГО ПРОГРАММИРОВАНИЯ

4.1. Регулярный фрактал "снежинка" Коха

Приводимая далее спецификация порождает регулярный фрактал "Снежинка Коха". Он получается из равностороннего треугольника следующим образом. Каждая из сторон треугольника, начиная с основания, разбивается на три равные части L_1, L_2 и L_3. На среднем отрезке L_2, как на основании, строится равносторонний треугольник, а крайние отрезки L_1 и L_3 дробятся аналогичным образом, и на каждой новой средней части (L_2) также строятся равносторонние треугольники. Дробление сторон вновь образуемых треугольников повторяется, пока размеры отрезков достаточно велики (для разрешения данного монитора), чтобы на них можно было построить визуально различимые треугольники.

Решение представляется в стиле объектно-синтаксического программирования в соответствии с мнемонической формулой:

программа = объекты + грамматика.

Экземпляры объектов создаются семантиками и терминал-действиями, реализуемыми методами объектов. Вызов методов управляется грамматикой в зависимости от текущего состояния операционной среды, тестируемого резольверами. Резольверы также реализуются методами (булевскими функциями) объектов.

4.2. Спецификация фрактала "снежинка" Коха

ФАЙЛ Koch.grm

При процессировании отключать трассировку. В противном случае в середине процесса наступит переполнение коллекции (окна трасировки).

После запуска процессирования "мышка" перестает быть активной. Так бывает, когда в спецификации операционной среды используется модуль DRIVERS (такова особенность DOS).

Данная спецификация определяет построение регулярного фрактала "Снежинка Коха" в стиле "ПРОГРАММА = ОБЪЕКТЫ + ГРАММАТИКА". Он получается из равностороннего треугольника следующим образом: каждая сторона треугольника делится на три равные части L1, L2, L3 и на средних отрезках (L2) как на основаниях строятся равносторонние треугольники меньших размеров. Крайние отрезки (L1 и L3) также делятся на три равные части и на средних частях, как на основаниях, опять строятся равносторонние треугольники. На новых треугольниках процесс повторяется до тех пор, пока позволяет разрешение монитора. Порождающий процессор, руководимый нижеприведенной грамматикой, рисует приближение фрактала следующим образом:

Обработка начинается по правилу B: исполняется терминал-действие 'DrawBase', задача которого нарисовать основание самого большого треугольника. Терминал-действие 'Open_L_in_B' создает экземпляр объекта L, поставляет ему координаты концов основания, и передает управление правилу для L. Семантики, вызываемые этим правилом, исполняются с данными этого экземпляра объекта L. Затем после того, как правило для L завершится, исполняется терминал-действие 'Close_L', уничтожающее экземпляр объекта, созданный терминалом 'Open_L_in_B'. Затем терминал-действие 'Open_T_in_B' создает экзем-

пляр объекта T для правила T, которое исполняется. Наконец, терминал-действие 'Close_T' уничтожает экземпляр T. На этом исполнение правила B заканчивается.

Рассмотрим, что происходит при выполнении правила T. Выполнение этого правила начинается с проверки параметров, полученных экземпляром объекта, созданного для этого правила. Эту проверку выполняет (полиморфный) резольвер resolution. Полученные параметры – координаты концов основания некоторого треугольника. Если расстояние между этими точками меньше, чем установленный предел разрешения монитора, то выполнение правила заканчивается. В противном случае, по координатам вершин треугольника (координаты третьей вершины уже были вычислены конструктором), терминал-действие 'DrawTriangle' рисует его боковые стороны.

Стоит отметить, что хотя наша картинка состоит из треугольников, полностью прорисовывается только один треугольник - самый большой, т.к. основания остальных сливаются с уже нарисованными линиями и таким образом их прорисовывать не требуется.

Далее выполняется терминал-действие 'Open1_L_in_T', которое создает экземпляр объекта L, посылает ему в качестве параметров координаты концов одной из боковых сторон треугольника, соответствующего текущему правилу T, и передает управление правилу L. После того, как исполнение правила L завершено, исполняется терминал-действие 'Close_L'. Терминал Open2_L_in_T, передает очередному создаваемому экземпляру объекта L координаты концов другой боковой стороны, снова исполняется правило L (но в другом окружении), и, наконец, терминал-действие 'Close_L' завершает исполнение правила T.

Выполнение правила L состоит из проверки полученных параметров относительно чувствительности монитора. Если расстояние между точками, координаты которых получены, меньше, чем установленный порог, то исполнение правила прекращается. В противном случае выполнятся терминал-действие 'Open1_L_in_L', правило L, терминал-действия 'Close_L', 'Open_T_in_L', правило T, терминал-действия 'Close_T', 'Open2_L_in_L', правило L, но для изменившейся операционной среды, и терминал-действие 'Close_L'.

Рассмотрим, каковы параметры, передаваемые терминалами 'Open1_L_in_L', 'Open_T_in_L' и 'Open2_L_in_L', создаваемым ими экземплярам соответствующих объектов. Когда начинает выполняться правило L, мы имеем координаты концов некоторого отрезка. Конструктор объекта, соответствующий правилу L, вычисляет координаты еще двух точек, лежащих на этом отрезке и делящих этот отрезок на три равные части. Обозначим подряд эти точки n1, n2, n3 и n4. Пусть L1=(n1,n2), L2=(n2,n3), L3 = (n3,n4). Терминал-действие 'Open1_L_in_L' передает координаты концов L1, терминал-действие 'Open_T_in_L' - координаты концов L2, а терминал-действие 'Open2_L_in_L' - координаты концов L3.

Порождающая управляющая грамматика

```
PRODUCTIVE SYNTAX
NONTERMINALS: S, B, T, L.
TERMINALS: 'Start','Finish','DrawBase','DrawTriangle',
           'Open_L_in_B', 'Open1_L_in_T', 'Open2_L_in_T',
           'Open1_L_in_L','Open2_L_in_L','Open_T_in_B',
           'Open_T_in_L','Close'.
FORWARD PASS RESOLVERS: resolution.
```

```
{ Правила: }
S:'Start', {Вычисление координат концов основания  базового
            треугольника и создание экземпляра B, представляющего
            его}
   B,        {Обработка по правилу B}
   'Finish'. {Уничтожение экземпляра объекта B}
B:'DrawBase',    {Построение основаения базового треугольника на
                  экране}
   'Open_L_in_B', {Создание экземпляра объекта L и
                   передача ему координат концов основания}
   L,             {Обработка по правилу L}
   'Close',       {Уничтожение экземпляра объекта L}
   'Open_T_in_B', {Создание экземпляра объекта T и
                   передача ему координат концов основания}
   T,             {Создание экземпляра объекта T}
   'Close'.       {Уничтожение экземпляра объекта T}
T:resolution,     {Проверка, допускают ли размеры фигуры нарисовать
                   ее на экране}
   'DrawTriangle', {Проведение боковых сторон треугольника}
   'Open1_L_in_T', {Создание экземпляра объекта L, представляющего
                    одну из проведенных боковых сторон треугольника}
   L ,             {Обработка по правилу L}
   'Close',        {Уничтожение экземпляра объекта L}
   'Open2_L_in_T', {Создание экземпляра объекта L, представляющего
                    другую из проведенных боковых сторон треугольника}
   L ,             {Обработка по правилу L}
   'Close';        {Уничтожение экземпляра объекта L}
   &.              {Пустое действие, если дальнейшее дробление
                    невозможно}
L: resolution,  {Проверка, допускают ли размеры фигуры нарисовать ее}
                 'Open1_L_in_L', {Создание экземпляра L и передача
                 ему координат концов левого отрезка}
   L ,           {Обработка по правилу L}
   'Close',      {Уничтожение экземпляра объекта L}
                 'Open_T_in_L', {Создание экземпляра объекта T и
                 передача ему координат концов среднего отрезка}
   T,            {Обработка по правилу T}
   'Close',      {Уничтожение экземпляра объекта T}
   'Open2_L_in_L', {Создание экземпляра объекта L и передача ему
                    координат концов правого отрезка}
   L ,           {Обработка по правилу L}
   'Close';      {Уничтожение экземпляра объекта L}
   &.
ENVIRONMENT
Uses Objects, Graph, Drivers, CRT;
{ Типы данных }
type { Точка на плоскости экрана }
 Point = record x, y : integer;
end;
```

```
{ Протообъект }
PProto = ^TProto;
TProto = object (TObject)
 Prev : PProto;
 n1, n2 : Point;
 destructor Done; virtual;
 procedure DrawBase; virtual;
 procedure DrawTriangle; virtual;
 procedure Open_L; virtual;
 procedure Open1_L; virtual;
 procedure Open2_L; virtual;
 procedure Open_T; virtual;
 procedure Close; virtual;
 function resolution : boolean; virtual;
end;
PB = ^TB;
TB = object( TProto )
{ Методы }
 constructor Init( v1, v2 : Point);
 procedure DrawBase; virtual;
 procedure Open_L; virtual;
 procedure Open_T; virtual;
end;
PT = ^TT;
TT = object(TProto)
 RESOL : integer;
 n3 : Point;
{ Методы }
 constructor Init (v1, v2 : Point);
 procedure DrawTriangle; virtual;
 procedure Open1_L; virtual;
 procedure Open2_L; virtual;
 function resolution: Boolean; virtual;
end;
PL = ^TL;
TL = object (TT) n4 : Point;
{ Методы }
 constructor Init( v1, v2 : Point );
 procedure Open1_L; virtual;
 procedure Open_T; virtual;
 procedure Open2_L; virtual;
 function resolution : Boolean; virtual;
end;
const CurrentEnvironment : PProto = Nil;
var RESOL : integer; {Мера минимальной допустимой величины рисуемых
                      элементов}
    nn1, nn2, nn3, nn4 : Point ;
    nX, nY : integer;
    grDriver, grMode, ErrCode : integer;
{ Реализация методов }
destructor TProto.Done; begin { void } end;
procedure TProto.DrawBase; begin abstract end;
procedure TProto.DrawTriangle; begin abstract end;
```

```
procedure TProto.Open_L; begin abstract end;
procedure TProto.Open1_L; begin abstract end;
procedure TProto.Open2_L; begin abstract end;
procedure TProto.Close;
var Tmp : PProto;
begin
 Tmp := Prev;
 Dispose( CurrentEnvironment, Done );
 CurrentEnvironment := Tmp
end;
procedure TProto.Open_T; begin abstract end;
function TProto.resolution : boolean; begin abstract end;
constructor TB.Init( v1, v2 : Point );
begin Prev := CurrentEnvironment; n1 := v1; n2 := v2 end;
procedure TB.DrawBase;
begin Line(n1.x, n1.y, n2.x, n2.y) end;
procedure TB.Open_L;
begin CurrentEnvironment := New (PL, Init(n2, n1)) end;
procedure TB.Open_T;
begin CurrentEnvironment := New ( PT, Init (n1, n2) ) end;
constructor TT.Init (v1, v2 : Point ) ;
var x,y : integer;
begin
 RESOL := 1;
 Prev := CurrentEnvironment; n1 := v1; n3 := v2;
{Вычисление кооридинат вершин треугольника с учетом направления
сторон}
 x := n3.x - n1.x; y := n3.y - n1.y;
 n2.x :=Round( x/2 - sqrt(3)*y/2 + n1.x);
 n2.y :=Round( y/2 + sqrt(3)*x/2 + n1.y)
end;
procedure TT.DrawTriangle;
var x,y : Integer;
begin { Построение боковых сторон (n1,n2) и (n2,n3) }
 Line(n1.x,n1.y, n2.x, n2.y);Line(n2.x, n2.y, n3.x, n3.y)
end;
procedure TT.Open1_L;
begin CurrentEnvironment := New( PL, Init (n1,n2) ) end;
procedure TT.Open2_L;
begin CurrentEnvironment := New( PL, Init (n2,n3) )end;
function TT.resolution : Boolean;
var x1, x2, y1, y2, d : real;
begin
 x1 := n1.x/10; x2 := n3.x/10;
 y1 := n1.y/10; y2 := n3.y/10;
 d := 10*sqrt( sqr( x2 - x1 ) + sqr( y2 - y1 ) );
 resolution := Round( d ) > RESOL
end;
constructor TL.Init( v1,v2 : Point);
begin
 RESOL:=3; Prev:=CurrentEnvironment; n1:=v1; n4:=v2;
 { Вычисление координат точек n2 и n3 для объекта L }
 n2.x :=n1.x + Round((n4.x - n1.x)/3);
 n2.y :=n1.y + Round((n4.y - n1.y)/3);
 n3.x :=n1.x + Round( 2*(n4.x - n1.x)/3);
 n3.y :=n1.y + Round( 2*(n4.y - n1.y)/3)
end;
```

```
procedure TL.Open1_L;
begin CurrentEnvironment := New( PL, Init (n1,n2) ) end;
procedure TL.Open_T;
begin CurrentEnvironment := New (PT, Init (n2,n3) ) end;
procedure TL.Open2_L;
begin CurrentEnvironment := New( PL, Init (n3,n4) ) end;
function TL.resolution : Boolean;
var x1, x2, y1, y2, d : real;
begin
x1 := n1.x/10; x2 := n4.x/10; y1 := n1.y/10; y2 := n4.y/10;
d := 10*sqrt( sqr( x2 - x1 ) + sqr( y2 - y1 ) ); resolution :=
Round( d ) > RESOL
end;
IMPLEMENTATION
{ Терминал-действия }
procedure Start;
begin
 grDriver := Detect;
 InitGraph(grDriver, grMode,'c:\BP\BGI');
 ErrCode := GraphResult;
 if ErrCode <> grOk
 then
 begin
  PrintString('Graphics error: '+GraphErrorMsg(ErrCode));
  exit
 end;
{ Устанавка графического режима }
 DoneVideo;
    {Вычисление координат концов
    основания базового треугольника}
 nX := GetMaxX; nY := GetMaxY;
 nn1.x := Round(3*nX/4); nn1.y := Round(2*nY/3);
 nn2.x := Round(nX/4); nn2.y := Round(2*nY/3);
{Создание первого экземпляра B}
 CurrentEnvironment := New( PB, Init(nn1, nn2) )
end;
procedure DrawBase;
{ Рисует основание первого треугольника }
begin CurrentEnvironment^.DrawBase end;
procedure DrawTriangle;
begin CurrentEnvironment^.DrawTriangle end;
procedure Open_L_in_B;
begin CurrentEnvironment^.Open_L end;
procedure Open_T_in_B;
begin CurrentEnvironment^.Open_T end;
procedure Open1_L_in_T;
begin CurrentEnvironment^.Open1_L end;
procedure Open2_L_in_T;
begin CurrentEnvironment^.Open2_L end;
```

```
procedure Open1_L_in_L;
begin CurrentEnvironment^.Open1_L end;
procedure Open_T_in_L;
begin CurrentEnvironment^.Open_T end;
procedure Open2_L_in_L;
begin CurrentEnvironment^.Open2_L end;
procedure Close;
begin CurrentEnvironment^.Close end;
procedure Finish;
begin
 {Уничтожить самый "первый" объект}
 Dispose (CurrentEnvironment, Done);
 SetTextStyle(DefaultFont, HorizDir, 2);
 OutTextXY (0, 18, '"СНЕЖИНКА" КОХА');
 { Задержать экран до нажатия любой клавиши }
 ReadKey;
 CloseGraph; InitVideo
end;
{ Резольверы }
function resolution : boolean;
begin resolution := CurrentEnvironment^.resolution end;
```

4.3. Управляющая граф-схема "снежинка" Коха

August 2, Thursday, 2001 10:52:57 0

0	Begin	S	20	T	'Close'
1	T	'Start'	21	T	'Open2_L_in_T'
2	N	B	22	N	L
3	T	'Finish'	23	T	'Close'
4	End	S	24	→	25
5	Begin	B	25	End	T
6	T	'DrawBase'	26	Begin	L
7	T	'Open_L_in_B'	27	–<	39
8	N	L	28	FR	resolution
9	T	'Close'	29	T	'Open1_L_in_L'
10	T	'Open_T_in_B'	30	N	L
11	N	T	31	T	'Close'
12	T	'Close'	32	T	'Open_T_in_L'
13	End	B	33	N	T
14	Begin	T	34	T	'Close'
15	–<	25	35	T	'Open2_L_in_L'
16	FR	resolution	36	N	L
17	T	'DrawTriangle'	37	T	'Close'
18	T	'Open1_L_in_T'	38	–<	39
19	N	L	39	End	L

Terminals :

```
1.'Start'           5.'Open_L_in_B'      9.'Open2_L_in_L'
2.'Finish'          6.'Open1_L_in_T'    10.'Open_T_in_B'
3.'DrawBase'        7.'Open2_L_in_T'    11.'Open_T_in_L'
4.'DrawTriangle'    8.'Open1_L_in_L'    12.'Close'
```

NonTerminals :

```
1.B ( B )   3.S ( S )
2.L ( L )   4.T ( T )
```

Forward Pass Resolvers :

```
1.resolution
```

4.4. Минимизированная управляющая таблица "снежинка" Коха

I. Таблица переходных состояний

Таблица П.1

Вход	Резольвер	Магазин	Состояние
1	**2**	**3**	**4**
Состояние 1={0}			
Start			2
Состояние 2={1}			
DrawBase		-22	3
Состояние 3={6}			
Open_L_in_B			4
Состояние 4={7}			
ε		-6	Suppress
Open1_L_in_L	resolution	-6	5
Состояние 5={29}			
ε		-7	Suppress
Open1_L_in_L	resolution	-7	6
Состояние 6={8}			
Close			8
Состояние 7={30}			
Close			9
Состояние 8={9}			
Open_T_in_B			10
Состояние 9={31}			
Open_T_in_L			11
Состояние 10={10}			
ε		-14	Suppress
DrawTriangle	resolution	-14	12

Продолжение табл. П.1

1	2	3	4
Состояние 11={32}			
ε		-15	Suppress
DrawTriangle	resolution	-15	12
Состояние 12={17}			
Open1_L_in_T			13
Состояние 13={18}			
ε		-16	Suppress
Open1_L_in_L	resolution	-16	5
Состояние 14={11}			
Close			17
Состояние 15={33}			
Close			18
Состояние 16={19}			
Close			19
Состояние 17={12}			
ε			Suppress
Состояние 18={34}			
Open2_L_in_L			20
Состояние 19={20}			
Open2_L_in_T			21
Состояние 20={35}			
ε		-23	Suppress
Open1_L_in_L	resolution	-23	5
Состояние 21={21}			
ε		-24	Suppress
Open1_L_in_L	resolution	-24	5
Состояние 22={2}			
Finish			25
Состояние 23={36}			
Close			26
Состояние 24={22}			
Close			27
Состояние 25={3} (конечное)			
ε			Stop
Состояние 26={37}			
ε			Suppress
Состояние 27={23}			
ε			Suppress

Замечание. Ради краткости в таблице П.1 вместо классов эквивалентных состояний указаны состояния. Их характеристики взяты из исходной управляющей таблицы.

4.5. Результат исполнения программы "снежинка" Коха

Результат исполнения программы, специфицированной в предыдущем параграфе, представлен на нижеследующем рисунке.

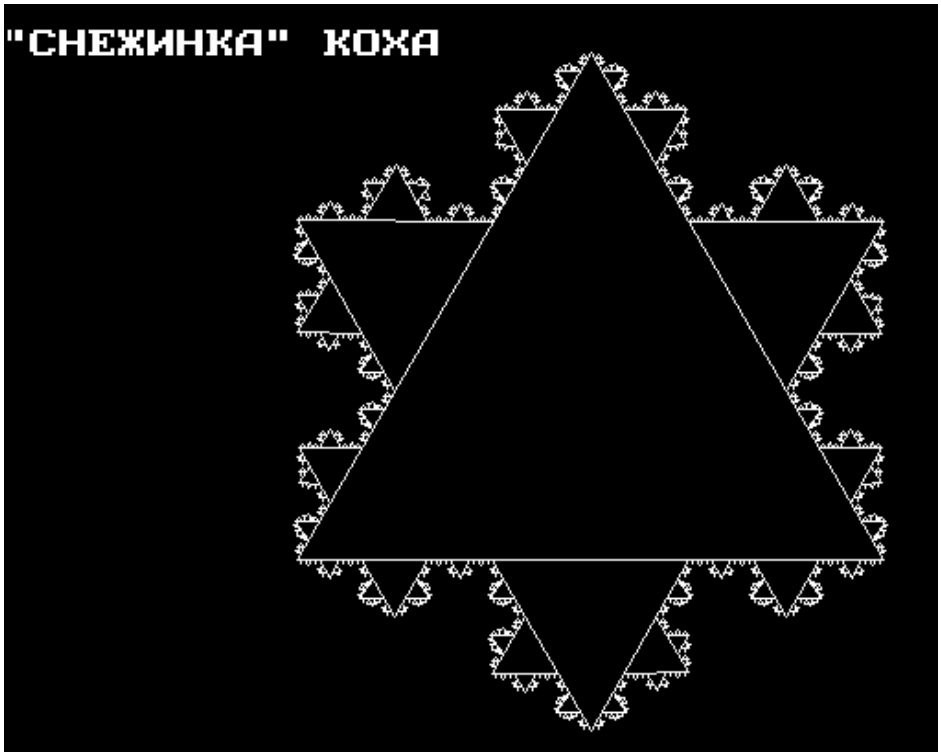


Рис. П.57. Регулярный фрактал "снежинка Коха".

Указатель литературы

1. **Алгол 68.** Методы реализации / Под ред. Г.С. Цейтина. Л.: Изд-во Ленингр. ун-та, 1976. 224 с.
2. **Ахо А., Ульман Дж.** Теория синтаксического анализа, перевода и компиляции. Т.1. Синтаксический анализ, 612 с.; Т.2. Компиляция, 487 с., М.: Мир, 1978.
3. **Майника Э.** Алгоритмы оптимизации на сетях и графах. М.: Мир, 1981. 323 с.
4. **Мартыненко Б.К., Косинец И.Э.** К обоснованной реализации языков программирования: построение и тестирование конечных процессоров. Деп. в ВИНИТИ, № 6909-84. 96 с.
5. **Мартыненко Б.К.** Технологический комплекс разработки языковых процессоров // Тр. V семинара "Проблемы информатики и ее применения в управлении, обучении и научных исследованиях". София, Соф. ун-т им. Клемента Охридского, 1988. С. 207–216.
6. **Мартыненко Б.К.** SYNTAX — инструментальный комплекс разработки языковых процессоров // Математическое моделирование и информационные технологии. Ижевск, 1991. С. 126–133.
7. **Пересмотренное** сообщение об АЛГОЛе 68. М.: Наука, 1979. 533 с.
8. **Харари Ф.** Теория графов. М.: Мир, 1973. 300 с.
9. **Chomsky N.** Three models for the description of language // IRE Trans. Inform. Theory. 1956. Vol.2, №3. P.113–124.
10. **Goodenough J.B., Gerhart S.L.** Toward a theory of testing: data selection criteria // Current trends in programming methodology. Vol.II. Program Validation / Ed. Yeh Raymond Tzuu-Yau. London, Printice-Hall, Inc, 1977. P. 44–79.
11. **Gray R.W., Heuring V.P., Levi S.P. e.a.** Eli: A complete, flexible compiler construction system // CACM. 1992. Vol.35, №2. P.121–130.
12. **Houssais B.** Verification of an Algol 68 implementation // SIGPLAN notices. 1977. Vol.12, №6. P.117–128.
13. **Johnson S.C.** Yacc — yet another compiler compiler // Comp. Sci. Tech. Rep. 32. Bell Telephone Laboratories. July, 1975. New Jersey: Murray Hill.
14. **Kastens U.** The GAG-system — a tool for compiler constrution // Methods and tools for compiler construction / Ed. Bernard Lorho. N.Y., Cambridge Univ. Press, 1984. P.165–182.
15. **Knuth D.E.** Semantics of context-free languages // Math. Systems Theory. 1968. Vol.2, №2, P.127–145.
16. **Lesk M.E.** LEX — a lexical analyzer generator // Comp. Sci. Tech. Rep.39. Bell Telephone Laboratories. Oct. 1975. New Jersey: Murray Hill.
17. **Miller E.F.** Program testing technology in the 1980's // Proc. Conf. Comp. 1980's. Portlend, Oregon: IEEE, 1978. P.117–128.
18. **Naur P.** Control-record-driven processing // Current Trends Programming Methodology. Data Structuring / Ed. Yeh Raymond Tzuu-Yau, London, Printice-Hall, Inc. 1977. Vol.IV. P.220 – 232.
19. **Turing A.M.** On computable numbers, with an application to the Entscheidungsproblem // Proc. London Math. Soc. 1936. Ser.2, Vol.42, P.230–265; Corrections. 1936, Vol.43. P.544–546.

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

Научное издание

Борис Константинович Мартыненко

Синтаксически управляемая обработка данных

Редактор *Т. В. Семенова*
Обложка художника *Е. А. Соловьевой*

Лицензия ИД № 05679 от 24.08.2001

Подписано в печать 27.12.2004. Формат 70x108 $1/_{16}$
Бумага офсетная. Печать офсетная.
Усл. печ. л. 27, 65. Заказ 45

Издательство СПбГУ. 199034, Санкт-Петербург, Университетская наб., 7/9

Тел. (812) 328-96-17; факс (812) 328-44-22

www.unipress.ru

По вопросам реализации обращаться по адресу:
С.-Петербург,6-я линия В.О. , д. 11/21,к. 21
Телефоны: 328-77-63, 325-31-76
E-mail: post@unipress.ru

Типография Издательства СПбГУ 199061, С.-Петербург, Средний пр., 41

В монографии описывается актуальная для практической информатики технология синтаксически управляемой обработки данных, использующая кусочно-регулярную аппроксимацию КС-языков. Трансляции специфицируются при помощи RBNF-грамматик и реализуются посредством контекстно чувствительных сплайновых языковых процессоров. Технология применяется для решения синтаксических проблем, а также поддерживает объектно-синтаксическую парадигму программирования, которая выражается метафорической формулой «программа = объекты + грамматика».

Книга предназначена для специалистов, работающих в области теории и технологии программирования, а также для студентов, изучающих информатику.

	Состояние 5
d	InitInt;SetDig
+	PushMonOp
-	PushMonOp
(	PushOpenPar
	Состояние 6
d	InitInt;SetDig
	Состояние 7
d	SetSign;InitInt;
+	SetSign
-	SetSign
	Состояние 12
)	AppDig;SetFrPart;AppFrPart;PushOpd

9 785288 036415

ИЗДАТЕЛЬСТВО САНКТ-ПЕТЕРБУРГСКОГО УНИВЕРСИТЕТА